

Sixth Symposium on Operating Systems Design and Implementation

San Francisco, CA, USA

December 6-8, 2004

Sponsored by
The USENIX Association

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

in cooperation with ACM SIGOPS

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
URL: <http://www.usenix.org>

The price is \$35 for members and \$45 for nonmembers.

Outside the U.S.A. and Canada, please add
\$15 per copy for postage (via air printed matter).

Past OSDI Proceedings

OSDI '02 (Fifth)	December 2002	Boston, Massachusetts, USA	\$25/32
OSDI '00 (Fourth)	October 2000	San Diego, California, USA	\$25/32
OSDI '99 (Third)	February 1999	New Orleans, Louisiana, USA	\$23/30
OSDI '96 (Second)	October 1996	Seattle, Washington, USA	\$20/27
OSDI '94 (First)	November 1994	Monterey, California, USA	\$20/27

© 2004 by The USENIX Association
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN 1-931971-26-9

USENIX Association

**Proceedings of the
Sixth Symposium on Operating Systems
Design and Implementation (OSDI '04)**

**December 6–8, 2004
San Francisco, CA, USA**

Symposium Organizers

Program Co-Chair

Eric Brewer, *University of California, Berkeley*

Peter Chen, *University of Michigan, Ann Arbor*

Program Committee

Miguel Castro, *Microsoft Research*

Jason Flinn, *University of Michigan*

Greg Ganger, *Carnegie Mellon University*

Samuel Madden, *Massachusetts Institute of Technology*

Jeff Mogul, *Hewlett Packard Labs*

Andrew Myers, *Cornell University*

Jason Nieh, *Columbia University*

Timothy Roscoe, *Intel Research*

Mendel Rosenblum, *Stanford University*

Margo Seltzer, *Harvard University*

Geoff Voelker, *University of California, San Diego*

David Wagner, *University of California, Berkeley*

Matt Welsh, *Harvard University*

Steering Committee

David Culler, *University of California, Berkeley*

Peter Druschel, *Rice University*

Mike Jones, *Microsoft Research*

The USENIX Association Staff

External Reviewers

Daniel Abadi

Sarita Adve

Atul Adya

David Andersen

Remzi Arpaci-Dusseau

Michael Bailey

Mary Baker

Krishnamurthy Balachander

Dirk Balfanz

Paul Barham

Murtaza Basrai

Christopher Batten

Ranjita Bhagwan

Sanjit Biswas

Richard Black

Chandrasekhar Boyapati

Scott Bradner

Thomas Bressoud

David Brooks

Aaron Brown

Angela Demke Brown

Kurt Brown

George Candea

Sirish Chandrasekaran

Fangzhe Chang

Fay Chang

Hao Chen

Tzi-cker Chiueh

Brent Chun

Michael Clarkson

Mark Corner

Manuel Costa

Landon Cox

Steven Czerwinski

Frank Dabek

Michael Demmer

Fred Douglass

Larry Dowdy

Peter Druschel

Bowei Du

Daniel Ellard

Carla Ellis

Jeremy Elson

Dawson Engler

Kevin Fall

Alexandra Fedorova

Hubertus Franke

Tal Garfinkel

David Gay

Jim Gettys

Phillip Gibbons

Thomer Gil

Richard Golding

Patrick Goldsack

Steven Gribble

Dan Grossman

Steven Hand

Paul Hargrove

Tim Harris

Hermann Härtig

John Heidemann

Joseph Hellerstein

David Holland

Anne Holler

Peter Honeyman

Wilson Hsieh

Bret Hull

Liviu Iftode

Arun Iyengar

Farnam Jahanian

Trevor Jim

William Josephson

Flavio Junqueira

Frans Kaashoek

Christos Karamanolis

Chris Karlof

Brad Karp

Jeff Kephart

Angelos Keromytis

Hyong-youb Kim

Minkyong Kim

Sam King

Evan Kirshenbaum

Eddie Kohler

Ramana Rao Kompella

Philip Koopman

Michael Kozuch

Orran Krieger

Geoff Kuenning

Monica Lam

Ed Lazowska

Jonathan Ledlie

Jay Lepreau

Philip Levis

Hank Levy

David Lie

Konrad Lorincz

David Lowell

Joshua MacDonald

Dan Magenheimer

Zhuoqing Morley Mao

Kostas Magoutis

Dave Maltz

Petros Maniatis

Keith Marzullo

William McCloskey

Ethan Miller

David Molnar

David Mosberger

Todd Mowry

Dushyanth Narayanan

Chris Newcombe

Brian Noble

Nathaniel Nystrom

Vivek Pai

Adrian Perrig

Ben Pfaff

Peter Pietzuch

Rob Pike

Venugopalan Ramasubramanian

Sylvia Ratnasamy

Peter Reiher

Mike Reiter

John Reumann

Sean Rhea

Martin Rinard

Scott Rixner

Rodrigo Rodrigues

Mema Roussopoulos

Ant Rowstron

Yasushi Saito

Naveen Sastry

Stefan Savage

Fred Schneider

Sebastian Schoenberg

David Schultz

Elizabeth Seamans

Russell Sears

R Sekar

Ken Sevcik

Hovav Shacham

Mehul Shah

Jonathan Shapiro

Marc Shapiro

Prashant Shenoy

Alex Sherman

Kang Shin

Emin Gun Sirer

Christopher Small

Allan Snaveley

Alex Snoeren

Dawn Song

Lex Stein

Dinesh Subhraveti

Sonesh Surana

Robert Szweczyk

Douglas Thain

Niraj Tolia

Richard Uhlig

Volkmar Uhlig

Amin Vahdat

Leendert van Doorn

Alistair Veitch

Mary Vernon

Giovanni Vigna

James von Behren

Jason Waddle

Carl Waldspurger

Dan Wallach

Kevin Walsh

Helen J. Wang

Yanling Wang

Brent Welch

Andrew Whitaker

Janet Wiener

John Wilkes

Robert Wisniewski

Alec Wolman

Theodore Wong

Jay Wylie

Haifeng Yu

Erez Zadok

Steve Zdancewicz

Yuanyuan Zhou

Feng Zhou

Lidong Zhou

**6th Symposium on Operating Systems
Design and Implementation (OSDI '04)
December 6–8, 2004
San Francisco, CA, USA**

Index of Authors vi

Message from the Symposium Chair vii

Monday, December 6, 2004

Dependability and Recovery

Recovering Device Drivers 1
Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy, University of Washington

Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines 17
Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz, University of Karlsruhe, Germany

Microreboot—A Technique for Cheap Recovery 31
George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox, Stanford University

Automated Management I

Automated Worm Fingerprinting 45
Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage, University of California, San Diego

Understanding and Dealing with Operator Mistakes in Internet Services 61
Kiran Nagaraja, Fábio Oliveira, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen, Rutgers University

Configuration Debugging as Search: Finding the Needle in the Haystack 77
Andrew Whitaker, Richard S. Cox, and Steven D. Gribble, University of Washington

File and Storage Systems I

Chain Replication for Supporting High Throughput and Availability 91
Robbert van Renesse and Fred B. Schneider, Cornell University

Boxwood: Abstractions as the Foundation for Storage Infrastructure 105
John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou, Microsoft Research Silicon Valley

Secure Untrusted Data Repository (SUNDR) 121
Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha, New York University

Distributed Systems

MapReduce: Simplified Data Processing on Large Clusters 137
Jeffrey Dean and Sanjay Ghemawat, Google, Inc.

FUSE: Lightweight Guaranteed Distributed Failure Notification 151
John Dunagan, Microsoft Research; Nicholas J. A. Harvey, Massachusetts Institute of Technology; Michael B. Jones, Microsoft Research; Dejan Kostić, Duke University; Marvin Theimer and Alec Wolman, Microsoft Research

PlanetSeer: Internet Path Failure Monitoring and Characterization in Wide-Area Services 167
Ming Zhang, Chi Zhang, Vivek Pai, Larry Peterson, and Randy Wang, Princeton University

Tuesday, December 7, 2004

Network Architecture

Improving the Reliability of Internet Paths with One-hop Source Routing 183
Krishna P. Gummadi, Harsha V. Madhyastha, Steven D. Gribble, Henry M. Levy, and David Wetherall, University of Washington

CoDNS: Improving DNS Performance and Reliability via Cooperative Lookups 199
KyoungSoo Park, Vivek S. Pai, Larry Peterson, and Zhe Wang, Princeton University

Middleboxes No Longer Considered Harmful 215
Michael Walfish, Jeremy Stribling, Maxwell Krohn, Hari Balakrishnan, and Robert Morris, MIT Computer Science and Artificial Intelligence Laboratory; Scott Shenker, University of California, Berkeley, and ICSI

Automated Management II

Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control .. 231
Ira Cohen, Hewlett-Packard Laboratories; Jeffrey S. Chase, Duke University; Moises Goldszmidt, Terence Kelly, and Julie Symons, Hewlett-Packard Laboratories

Automatic Misconfiguration Troubleshooting with PeerPressure 245
Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang, Microsoft Research

Using Magpie for Request Extraction and Workload Modelling 259
Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier, Microsoft Research, Cambridge, UK

Bugs

Using Model Checking to Find Serious File System Errors 273
Junfeng Yang, Paul Twohey, and Dawson Engler, Stanford University; Madanlal Musuvathi, Microsoft Research

CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code 289
Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou, University of Illinois at Urbana-Champaign

Enhancing Server Availability and Security Through Failure-Oblivious Computing 303
Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebe, Jr., Massachusetts Institute of Technology

Index of Authors

Arpaci-Dusseau, Andrea C.	317, 379	Martin, Richard P.	61
Arpaci-Dusseau, Remzi H.	317, 379	Mazières, David	121
Annamalai, Muthukaruppan	1	Mortier, Richard	259
Bairavasundaram, Lakshmi N.	379	Morris, Robert	215
Balakrishnan, Hari	215	Murphy, Nick	105
Barham, Paul	259	Musuvathi, Madanlal	273
Beebee Jr., William S.	303	Myagmar, Suvda	289
Bershad, Brian N.	1	Nagaraja, Kiran	61
Bianchini, Ricardo	61	Nahum, Erich	333
Bos, Herbert	347	Najork, Marc	105
de Bruijn, Willem	347	Nguyen, Thu D.	61
Butt, Ali R.	395	Nguyen, Trung	347
Cadar, Cristian	303	Nieh, Jason	333
Candea, George	31	Nightingale, Edmund B.	363
Chase, Jeffrey S.	231	Oliveira, Fábio	61
Chen, Yu	245	Olsheski, David P.	333
Cohen, Ira	231	Pai, Vivek	167, 199
Cox, Richard S.	77	Park, KyoungSoo	199
Cristea, Mihai	347	Peterson, Larry	167, 199
Dean, Jeffrey	137	Platt, John C.	245
Donnelly, Austin	259	Portokalidis, Georgios	347
Dumitran, Daniel	303	Van Renesse, Robert	91
Dunagan, John	151	Rinard, Martin	303
Engler, Dawson	273	Roy, Daniel M.	303
Estan, Cristian	45	Savage, Stefan	45
Flinn, Jason	363	Schneider, Fred B.	91
Fox, Armando	31	Shasha, Dennis	121
Friedman, Greg	31	Shenker, Scott	215
Fujiki, Yuichi	31	Singh, Sumeet	45
Ghemawat, Sanjay	137	Sivathanu, Muthian	379
Gniady, Chris	395	Stoess, Jan	17
Goldszmidt, Moises	231	Stribling, Jeremy	215
Götz, Stefan	17	Swift, Michael M.	1
Gribble, Steven D.	77, 183	Symons, Julie	231
Gummadi, Krishna P.	183	Theimer, Marvin	151
Gunawi, Haryadi S.	317	Thekkath, Chandramohan A.	105
Harvey, Nicholas J. A.	151	Twohey, Paul	273
Hu, Y. Charlie	395	Uhlig, Volkmar	17
Isaacs, Rebecca	259	Varghese, George	45
Jones, Michael B.	151	Walfish, Michael	215
Kawamoto, Shinichi	31	Wang, Helen J.	245
Kelly, Terence	231	Wang, Randy	167
Kostić, Dejan	151	Wang, Zhe	199
Krohn, Maxwell	121	Wetherall, David	183
LeVasseur, Joshua	17	Whitaker, Andrew	77
Leiden, Universiteit	347	Wolman, Alec	151
Leu, Tudor	303	Yang, Junfeng	273
Levy, Henry M.	1, 183	Yi-Wang, Min	245
Li, Jinyuan	121	Zhang, Chi	167
Li, Zhenmin	289	Zhang, Ming	167
Lu, Shan	289	Zhang, Ruyun	245
MacCormick, John	105	Zhou, Lidong	105
Madhyastha, Harsha V.	183	Zhou, Yuanyuan	289

Wednesday, December 8, 2004

Kernel Networking

Deploying Safe User-Level Network Services with icTCP 317
Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, University of Wisconsin, Madison

ksniffer: Determining the Remote Client Perceived Response Time from Live Packet Streams 333
David P. Olshefski, Columbia University and IBM T.J. Watson Research; Jason Nieh, Columbia University; Erich Nahum, IBM T.J. Watson Research

FFPF: Fairly Fast Packet Filters 347
Herbert Bos and Willem de Bruijn, Vrije Universiteit Amsterdam, The Netherlands; Mihai Cristea, Trung Nguyen, and Georgios Portokalidis, Universiteit Leiden, The Netherlands

File and Storage Systems II

Energy-Efficiency and Storage Flexibility in the Blue File System 363
Edmund B. Nightingale and Jason Flinn, University of Michigan

Life or Death at Block-Level 379
Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, University of Wisconsin, Madison

Program-Counter-Based Pattern Classification in Buffer Caching 395
Chris Gniady, Ali R. Butt, and Y. Charlie Hu, Purdue University

Message from the Program Chairs

This year marks the ten-year anniversary of OSDI. In the short time since OSDI began, it has become (along with its odd-year counterpart, SOSOP) the top conference in the systems software area. This year's OSDI continues the conference's tradition of publishing innovative and high-quality research from a broad set of areas, including dependability and recovery, automated management, file and storage systems, distributed systems, networks, security, power management, and debugging. For many of us, it is this tradition of quality from a wide range of relevant areas that keeps us coming back to OSDI.

To sustain this tradition, we recruited a committee that mirrored our goal for the program: outstanding researchers from a wide range of relevant areas. Each program committee member devoted an enormous amount of time and energy to the conference, with each member reviewing an average of 42 papers. We greatly appreciate their dedication and hard work.

We received 193 submissions to this year's OSDI—29% more than ever before. Not only was the submission pool unusually large, it was also of exceptionally high quality. Several of the program committee members remarked that the average quality of submissions in this batch was the best they had seen. All submissions were reviewed by several members of the program committee and selected external reviewers. Altogether we received nearly 900 reviews, many of them quite lengthy. Based on these reviews, the program committee selected 27 papers to be presented at the conference. Each accepted paper was shepherded by a member of the program committee. We believe the resulting papers represent some of the most innovative and exciting work being done in the field.

This conference is the result of the hard work of many. First and foremost, we want to thank each author who submitted a paper, including those whose papers we were not able to accept. Next, we want to thank the members of the program committee for the long hours spent reviewing 40+ papers each. We also want to thank the numerous external reviewers, many of whom reviewed papers on very short notice. Finally, we would like to thank the USENIX staff for their expert guidance and hard work in putting the conference together.

Eric Brewer, University of California, Berkeley
Peter Chen, University of Michigan, Ann Arbor
Symposium Co-Chairs

Recovering Device Drivers

Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy

Department of Computer Science and Engineering

University of Washington

Seattle, WA 98195 USA

`{mikesw, muthu, bershad, levy}@cs.washington.edu`

Abstract

This paper presents a new mechanism that enables applications to run correctly when device drivers fail. Because device drivers are the principal failing component in most systems, reducing driver-induced failures greatly improves overall reliability. Earlier work has shown that an operating system can survive driver failures [33], but the applications that depend on them cannot. Thus, while operating system reliability was greatly improved, application reliability generally was not.

To remedy this situation, we introduce a new operating system mechanism called a *shadow driver*. A shadow driver monitors device drivers and transparently recovers from driver failures. Moreover, it assumes the role of the failed driver during recovery. In this way, applications using the failed driver, as well as the kernel itself, continue to function as expected.

We implemented shadow drivers for the Linux operating system and tested them on over a dozen device drivers. Our results show that applications and the OS can indeed survive the failure of a variety of device drivers. Moreover, shadow drivers impose minimal performance overhead. Lastly, they can be introduced with only modest changes to the OS kernel and with no changes at all to existing device drivers.

1 Introduction

Improving reliability is one of the greatest challenges for commodity operating systems. System failures are commonplace and costly across all domains: in the home, in the server room, and in embedded systems, where the existence of the OS itself is invisible. At the low end, failures lead to user frustration and lost sales. At the high end, an hour of downtime from a system failure can result in losses in the millions [16].

Most of these system failures are caused by the operating system's device drivers. Failed drivers cause 85% of Windows XP crashes [30], while Linux drivers have seven times the bug rate of other kernel code [14]. A failed driver typically causes the application, the OS kernel, or both to crash or stop functioning as expected. Hence, preventing driver-induced failures improves overall system reliability.

Earlier failure-isolation systems within the kernel were designed to prevent driver failures from corrupting the kernel itself [33]. In these systems, the kernel unloads a failed driver and then restarts it from a safe initial state. While isolation techniques can reduce the frequency of system crashes, *applications* using the failed driver can still crash. These failures occur because the driver loses application state when it restarts, causing applications to receive erroneous results. Most applications are unprepared to cope with this. Rather, they reflect the conventional failure model: drivers and the operating system either fail together or not at all.

This paper presents a new mechanism, called a *shadow driver*, that improves overall system reliability by concealing a driver's failure from its clients while recovering from the failure. During normal operation, the shadow tracks the state of the real driver by monitoring all communication between the kernel and the driver. When a failure occurs, the shadow inserts itself *temporarily* in place of the failed driver, servicing requests on its behalf. While shielding the kernel and applications from the failure, the shadow driver restores the failed driver to a state where it can resume processing requests.

Our design for shadow drivers reflects four principles:

1. *Device driver failures should be concealed from the driver's clients.* If the operating system and applications using a driver cannot detect that it has failed, they are unlikely to fail themselves.
2. *Recovery logic should be centralized in a single subsystem.* We want to consolidate recovery knowledge in a small number of components to simplify the implementation.
3. *Driver recovery logic should be generic.* The increased reliability offered by driver recovery should not be offset by potentially destabilizing changes to the tens of thousands of existing drivers. Therefore, the architecture must enable a single shadow driver to handle recovery for a large number of device drivers.

4. *Recovery services should have low overhead when not needed.* The recovery system should impose relatively little overhead for the common case (that is, when drivers are operating normally).

Overall, these design principles are intended to minimize the cost required to make and use shadow drivers while maximizing their value in existing commodity operating systems.

We implemented the shadow driver architecture for sound, network, and IDE storage drivers on a version of the Linux operating system. Our results show that shadow drivers: (1) mask device driver failures from applications, allowing applications to run normally during and after a driver failure, (2) impose minimal performance overhead, (3) require no changes to existing applications and device drivers, and (4) integrate easily into an existing operating system.

This paper describes the design, implementation and performance of shadow drivers. The following section reviews general approaches to protecting applications from system faults. Section 3 describes device drivers and the shadow driver design and components. Section 4 presents the structure of shadow drivers and the mechanisms required to implement them in Linux. Section 5 presents experiments that evaluate the performance, effectiveness, and complexity of shadow drivers. The final section summarizes our work.

2 Related Work

This section describes previous research on recovery strategies and mechanisms. The importance of recovery has long been known in the database community, where transactions [19] prevent data corruption and allow applications to manage failure. More recently, the need for failure recovery has moved from specialized applications and systems to the more general arena of commodity systems [28].

A general approach to recovery is to run application replicas on two machines, a primary and a backup. All inputs to the primary are mirrored to the backup. After a failure of the primary, the backup machine takes over to provide service. The replication can be performed by the hardware [21], at the hardware-software interface [8], at the system call interface [2, 5, 7], or at a message passing or application interface [4]. Shadow drivers similarly replicate all communication between the kernel and device driver (the primary), sending copies to the shadow driver (the backup). If the driver fails, the shadow takes over temporarily until the driver recovers. However, shadows differ from typical replication schemes in several ways. First, because our goal is to tolerate only driver failures, not hardware failures, both the shadow and the “real” driver run on the same machine. Second,

and more importantly, the shadow is *not* a replica of the device driver: it implements only the services needed to manage recovery of the failed driver and to shield applications from the recovery. For this reason, the shadow is typically much simpler than the driver it shadows.

Another common recovery approach is to restart applications after a failure. Many systems periodically checkpoint application state [26, 27, 29], while others combine checkpoints with logs [2, 5, 31]. These systems transparently restart failed applications from their last checkpoint (possibly on another machine) and replay the log if one is present. Shadow drivers take a similar approach by replaying a log of requests made to drivers. Recent work has shown that this approach is limited when recovering from *application* faults: applications often become corrupted before they fail; hence, their logs or checkpoints may also be corrupted [10, 25]. Shadow drivers reduce this potential by logging only a small subset of requests. Furthermore, application bugs tend to be deterministic and recur after the application is restarted [11]. Driver faults, in contrast, often cause transient failures because of the complexities of the kernel execution environment [34].

Another approach is simply to reboot the failed component, for example, unloading and reloading failed kernel extensions, such as device drivers [33]. Rebooting has been proposed as a general strategy for building high-availability software [9]. However, rebooting forces *applications* to handle the failure, for example, reinitializing state that has been lost by the rebooted component. Few existing applications do this [9], and those that do not share the fate of the failed driver. Shadow drivers transparently restore driver state lost in the reboot, invisibly to applications.

Shadow drivers rely on device driver isolation to prevent failed drivers from corrupting the OS or applications. Isolation can be provided in various ways. Vino [32] encapsulates extensions using software fault isolation [35] and uses transactions to repair kernel state after a failure. Nooks [33] and Palladium [13] isolate extensions in protection domains enforced by virtual memory hardware. Microkernels [23, 38, 39] and their derivatives [15, 17, 20] force isolation by executing extensions in user mode.

Rather than concealing driver failures, these systems all reflect a *revealing* strategy, one in which the application or user is made aware of the failure. The OS typically returns an error code, telling the application that a system call failed, but little else (e.g., it does not indicate which component failed or how the failure occurred). The burden of recovery then rests on the application, which must decide what steps to take to continue executing. As previously mentioned, most applications cannot handle the failure of device drivers [37], since driver faults typ-

ically crash the system. When a driver failure occurs, these systems expose the failure to the application, which may then fail. By impersonating device drivers during recovery, shadow drivers conceal errors caused by driver failures and thereby protect applications.

Several systems have narrowed the scope of recovery to focus on a specific subsystem or component. For example, the Rio file cache [12] provides high performance by isolating a single system component, the file cache, from kernel failures. Phoenix [3] provides transparent recovery after the failure of a single problematic component type, database connections in multi-tier applications. Similarly, our shadow driver research focuses on recovery for a single OS component type, the device driver, which is the leading cause of OS failure. By abandoning general-purpose recovery, we transparently resolve a major cause of application and OS failure while maintaining a low runtime overhead.

3 Device Drivers and Shadow Driver Design

A device driver is a kernel-mode software component that provides an interface between the OS and a hardware device¹. The driver converts requests from the kernel into requests to the hardware. Drivers rely on two interfaces: the interface that drivers *export* to the kernel that provides access to the device, and the kernel interface that drivers *import* from the operating system. For example, Figure 1 shows the kernel calling into a sound driver to play a tone; in response, the sound driver converts the request into a sequence of I/O instructions that direct the sound card to emit sound.

In practice, most device drivers are members of a *class*, which is defined by its interface. For example, all network drivers obey the same kernel-driver interface, and all sound-card drivers obey the same kernel-driver interface. This class orientation simplifies the introduction of new drivers into the operating system, since no OS changes are required to accommodate them.

In addition to processing I/O requests, drivers also handle configuration requests. Applications may configure the device, for example, by setting the bandwidth of a network card or the volume for a sound card. Configuration requests may change both driver and device behavior for future I/O requests.

3.1 Driver Faults

Most drivers fail due to bugs that result from unexpected inputs or events [34]. For example, a driver may corrupt a data structure if an interrupt arrives during a sensitive

¹This paper uses the terms “device driver” and “driver” interchangeably; similarly, we use the terms “shadow driver” and “shadow” interchangeably.

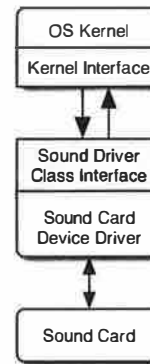


Figure 1: A sample device driver. The device driver *exports* the services defined by the device’s class interface and *imports* services from the kernel’s interface.

portion of request processing. Device drivers may crash in response to (1) the stream of requests from the kernel, both configuration and I/O, (2) messages to and from the device, and (3) the kernel environment, which may raise or lower power states, swap pages of memory, and interrupt the driver at arbitrary times. A driver bug triggered solely by a sequence of configuration or I/O requests is called a *deterministic* failure. No generic recovery technique can transparently recover from this type of bug, because any attempt to complete an offending request may trigger the bug [11]. In contrast, *transient* failures are triggered by additional inputs from the device or the operating system and occur infrequently.

A driver failure that is detected and stopped by the system before any OS, device, or application state is affected is termed *fail-stop*. More insidious failures may corrupt the system or application and never be detected. The system’s response to failure determines whether a failure is fail-stop. For example, a system that detects and prevents accidental writes to kernel data structures exhibits fail-stop behavior for such a bug, whereas one that allows corruption does not.

Appropriate OS techniques can ensure that drivers execute in a fail-stop fashion [32, 33, 36]. For example, in earlier work we described Nooks [33], a kernel reliability subsystem that executes each driver within its own in-kernel protection domain. Nooks detects faults through memory protection violations, excessive CPU usage, and certain bad parameters passed to the kernel. When Nooks detects a failure, it stops execution within the driver’s protection domain and triggers a recovery process. We reported that Nooks was able to detect approximately 75% of failures in synthetic fault-injection tests [33].

Shadow drivers can recover only from failures that are both transient and fail-stop. Deterministic failures may recur when the driver recovers, again causing a failure.

In contrast, transient failures are triggered by environmental factors that are unlikely to persist during recovery. In practice, many drivers experience transient failures, caused by the complexities of the kernel execution environment (e.g. asynchrony, interrupts, locking protocols, and virtual memory) [1], which are difficult to find and fix. Deterministic driver failures, in contrast, are more easily found and fixed in the testing phase of development because the failures are repeatable [18]. Recoverable failures must also be fail-stop, because shadow drivers *conceal* failures from the system and applications. Hence, shadow drivers require a reliability subsystem to detect and stop failures before they are visible to applications or the operating system. Although shadow drivers may use any mechanism that provides these services, our implementation uses Nooks.

3.2 Shadow Drivers

A *shadow driver* is a kernel agent that improves reliability for a single device driver. It compensates for and recovers from a driver that has failed. When a driver fails, its shadow restores the driver to a functioning state in which it can process I/O requests made before the failure. While the driver recovers, the shadow driver services its requests.

Shadow drivers execute in one of two modes: passive or active. In *passive* mode, used during normal (non-faulting) operation, the shadow driver monitors all communication between the kernel and the device driver it shadows. This monitoring is achieved via replicated procedure calls: a kernel call to a device driver function causes an automatic, identical call to a corresponding shadow driver function. Similarly, a driver call to a kernel function causes an automatic, identical call to a corresponding shadow driver function. These passive-mode calls are transparent to the device driver and the kernel. They are not intended to provide any service to either party and exist only to track the state of the driver as necessary for recovery.

In *active* mode, which occurs during recovery from a failure, the shadow driver performs two functions. First, it “impersonates” the failed driver, intercepting and responding to calls from the kernel. Therefore, the kernel and higher-level applications continue operating in as normal a fashion as possible. Second, the shadow driver impersonates the kernel to restart the failed driver, intercepting and responding to calls from the restarted driver to the kernel. In other words, in active mode the shadow driver looks like the kernel to the driver and like the driver to the kernel. Only the shadow driver is aware of the deception. This approach hides recovery details from the driver, which is unaware that it is being restarted by a shadow driver after a failure.

Once the driver has restarted, the active-mode shadow reintegrates the driver into the system. It re-establishes any application configuration state downloaded into the driver and then resumes pending requests.

A shadow driver is a “class driver,” aware of the interface to the drivers it shadows but *not* of their implementations. A single shadow driver implementation can recover from a failure of any driver in the class. The class orientation has three key implications. First, an operating system can leverage a few implementations of shadow drivers to recover from failures in a large number of device drivers. Second, implementing a shadow driver does not require a detailed understanding of the internals of the drivers it shadows. Rather, it requires only an understanding of those drivers’ interactions with the kernel. Finally, if a new driver is loaded into the kernel, no new shadow driver is required *as long as* a shadow for that class already exists. For example, if a new network interface card and driver are inserted into a PC, the existing network shadow driver can shadow the new driver without change. Similarly, drivers can be patched or updated without requiring changes to their shadows. Shadow updating is required only to respond to a change in the kernel-driver programming interface.

3.3 Taps

As we have seen, a shadow driver monitors communication between a functioning driver and the kernel and impersonates one component to the other during failure and recovery. These activities are made possible by a new mechanism, called a *tap*. Conceptually, a tap is a T-junction placed between the kernel and its drivers. It can be set to replicate calls during passive mode and redirect them during recovery.

A tap operates in passive or active mode, corresponding to the state of the shadow driver attached to it. During passive-mode operation, the tap: (1) invokes the original driver, then (2) invokes the shadow driver with the parameters and results of the call. This operation is shown in Figure 2.

On failure, the tap switches to active mode, shown in Figure 3. In this mode, it: (1) terminates all communication between the driver and kernel, and (2) redirects all invocations to their corresponding interface in the shadow. In active mode, both the kernel and the recovering device driver interact only with the shadow driver. Following recovery, the tap returns to its passive-mode state.

Taps depend on the ability to dynamically dispatch all communication between the driver and the OS. Consequently, all communication into and out of a driver being shadowed must be explicit, such as through a procedure call or a message. Most drivers operate this way, but some do not and cannot be shadowed. For example,

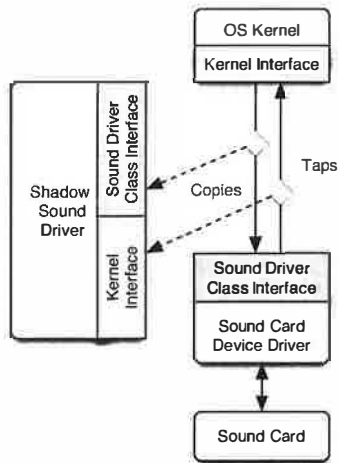


Figure 2: A sample shadow driver operating in passive mode. Taps inserted between the kernel and sound driver ensure that all communication between the two is passively monitored by the shadow driver.

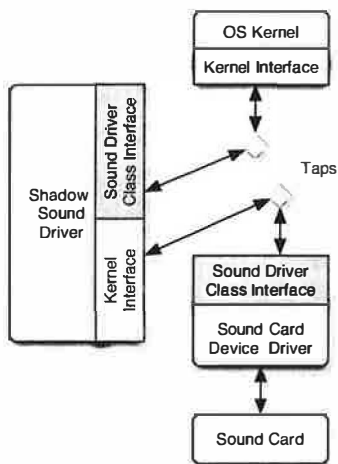


Figure 3: A sample shadow driver operating in active mode. The taps redirect communication between the kernel and the failed driver directly to the shadow driver.

kernel video drivers often communicate with usermode applications through shared memory regions [22].

3.4 The Shadow Manager

Recovery is supervised by the *shadow manager*, which is a kernel agent that interfaces with and controls all shadow drivers. The shadow manager instantiates new shadow drivers and injects taps into the call interfaces between the device driver and kernel. It also receives notification from the fault-isolation subsystem that a driver has stopped due to a failure.

When a driver fails, the shadow manager transitions its taps and shadow driver to active mode. In this mode,

requests for the driver's services are redirected to an appropriately prepared shadow driver. The shadow manager then initiates the shadow driver's recovery sequence to restore the driver. When recovery ends, the shadow manager returns the shadow driver and taps to passive-mode operation so the driver can resume service.

3.5 Summary

Our design simplifies the development and integration of shadow drivers into existing systems. Each shadow driver is a single module written with knowledge of the behavior (interface) of a class of device drivers, allowing it to conceal a driver failure and restart the driver after a fault. A shadow driver, normally passive, monitors communication between the kernel and the driver. It becomes an active proxy when a driver fails and then manages its recovery.

4 Shadow Driver Implementation

This section describes the implementation of shadow drivers in the Linux operating system [6]. We have implemented shadow drivers for three classes of device drivers: sound card drivers, network interface drivers, and IDE storage drivers.

4.1 General Infrastructure

All shadow drivers rely on a generic service infrastructure that provides three functions. An *isolation service* prevents driver errors from corrupting the kernel by stopping a driver on detecting a failure. A transparent *redirection mechanism* implements the taps required for transparent shadowing and recovery. Lastly, an *object tracking service* tracks kernel resources created or held by the driver so as to facilitate recovery.

Our shadow driver implementation uses Nooks to provide these functions. Through its fault isolation subsystem, Nooks [33] isolates drivers within separate kernel protection domains. The domains use memory protection to trap driver faults and ensure the integrity of kernel memory. Nooks interposes proxy procedures on all communication between the device driver and kernel. We insert our tap code into these Nooks proxies to replicate and redirect communication. Finally, Nooks tracks kernel objects used by drivers to perform garbage collection of kernel resources during recovery.

Our implementation adds a shadow manager to the Linux operating system. In addition to receiving failure notifications from Nooks, the shadow manager handles the initial installation of shadow drivers. In coordination with the kernel's module loader, which provides the driver's class, the shadow manager creates a new shadow driver instance for a driver. Because a single shadow driver services a class of device drivers, there may be

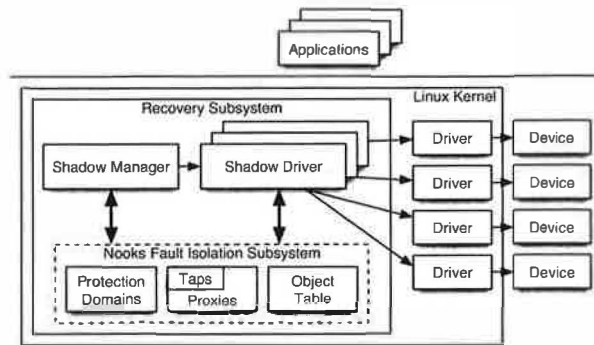


Figure 4: The Linux operating system with several device drivers and the driver recovery subsystem. New code components include the taps, the shadow manager and a set of shadow drivers, all built on top of the Nooks driver fault isolation subsystem.

several instances of a shadow driver executing if there is more than one driver of a class present. The new instance shares the same code with all other instances of that shadow driver class.

Figure 4 shows the driver recovery subsystem, which contains the Nooks fault isolation subsystem, the shadow manager, and a set of shadow drivers, each of which can monitor one or more device drivers.

4.2 Passive-Mode Monitoring

In passive mode, a shadow driver records several types of information. First, it tracks requests made to the driver, enabling pending requests to execute correctly after recovery. For connection-oriented drivers, the shadow driver records the state of each active connection, such as offset or positioning information. For request-oriented drivers, the shadow driver maintains a *log* of pending commands and arguments. An entry remains in the log until the corresponding request has been handled.

The shadow driver also records configuration and driver parameters that the kernel passes into the driver. During recovery, the shadow uses this information to act in the driver's place, returning the same information that was passed in previously. This information also assists in reconfiguring the driver to its pre-failure state when it is restarted. For example, the shadow sound driver keeps a log of `ioctl` calls (command numbers and arguments) that configure the driver. This log makes it possible to: (1) act as the device driver by remembering the sound formats it supports, and (2) recover the driver by resetting properties, such as the volume and sound format in use.

The shadow driver maintains only the *configuration* of the driver in its log. For stateful devices, such as frame buffers or storage devices, it does not create a copy of the

device state. Instead, a shadow driver depends on the fail-stop assumption to preserve persistent state (e.g., on disk) from corruption. It can restore transient state (state that is lost when the device resets) if it can force the device's clients to recreate that state, for example, by redrawing the contents of a frame buffer.

Lastly, the shadow tracks all kernel objects that the driver allocated or received from the kernel. These objects would otherwise be lost when the driver fails, causing a memory leak. For example, the shadow must record all timer callbacks registered and all hardware resources owned, such as interrupt lines and I/O memory regions.

In many cases, passive-mode calls do no work and the shadow returns immediately to the caller. For example, the dominant calls to a sound-card driver are `read` and `write`, which record or play sound. In passive mode, the shadow driver implements these calls as *no-ops*, since there is no need to copy the real-time sound data flowing through the device driver. For an `ioctl` call, however, the sound-card shadow driver logs the command and data for the connection. Similarly, the shadow driver for an IDE disk does little or no work in passive mode, since the kernel and disk driver handle all I/O and request queuing. Finally, for the network shadow driver, much of the work is already performed by the Nooks object-tracking system, which keeps references to outstanding packets.

4.3 Active-Mode Recovery

A driver typically fails by generating an illegal memory reference or passing an invalid parameter across a kernel interface. The kernel-level failure detector notices the failure and invokes the shadow manager, which locates the appropriate shadow driver and directs it to recover the failed driver. The three steps of recovery are: (1) stopping the failed driver, (2) reinitializing the driver from a clean state, and (3) transferring relevant shadow driver state into the new driver.

4.3.1 Stopping the Failed Driver

The shadow manager begins recovery by informing the responsible shadow driver that a failure has occurred. It also switches the taps, isolating the kernel and driver from one another's subsequent activity during recovery. After this point, the tap redirects all kernel requests to the shadow until recovery is complete.

Informed of the failure, the shadow driver first disables execution of the failed driver. It also disables the hardware device to prevent it from interfering with the OS while not under driver control. For example, the shadow disables the driver's interrupt request line. Otherwise, the device may continuously interrupt the kernel and prevent recovery. On hardware platforms with I/O memory mapping, the shadow also removes the device's I/O mappings

to prevent DMAs into kernel memory.

To prepare for restarting the device driver, the shadow garbage collects resources held by the driver. It retains objects that the kernel uses to request driver services, to ensure that the kernel does not see the driver “disappear” as it is restarted. The shadow releases the remaining resources.

4.3.2 Reinitializing the Driver

The shadow driver next “reboots” the driver from a clean state. Normally, restarting a driver requires reloading the driver from disk. However, we cannot assume that the disk is functional during recovery. For this reason, when creating a new shadow driver instance, the shadow manager caches in the shadow instance a copy of the device driver’s initial, clean data section. These sections tend to be small. The driver’s code is kernel-read-only, so it is not cached and can be reused from memory.

The shadow restarts the driver by initializing the driver’s state and then repeating the kernel’s driver initialization sequence. For some driver classes, such as sound card drivers, this consists of a single call into the driver’s initialization routine. Other drivers, such as network interface drivers, require additional calls to connect the driver into the network stack.

As the driver restarts, the shadow reattaches the driver to its pre-failure kernel resources. During driver reboot, the driver makes a number of calls into the kernel to discover information about itself and to link itself into the kernel. For example, the driver calls the kernel to register itself as a driver and to request hardware and kernel resources. The taps redirect these calls to the shadow driver, which reconnects the driver to existing kernel data structures. Thus, when the driver attempts to register with the kernel, the shadow intercepts the call and reuses the existing driver registration, avoiding the allocation of a new one. For requests that generate callbacks, such as a request to register the driver with the PCI subsystem, the shadow emulates the kernel, making the same callbacks to the driver with the same parameters. The driver also acquires hardware resources. If these resources were previously disabled at the first step of recovery, the shadow re-enables them, e.g., enabling interrupt handling for the device’s interrupt line. In essence, the shadow driver initializes the recovering driver by calling and responding as the kernel would when the driver starts normally.

4.3.3 Transferring State to the New Driver

The final recovery step restores the driver state that existed at the time of the fault, permitting it to respond to requests as if it had never failed. Thus, any configuration that either the kernel or an application had downloaded

to the driver must be restored. The details of this final state transfer depend on the device driver class. Some drivers are connection oriented. For these, the state consists of the state of the connections before the failure. The shadow re-opens the connections and restores the state of each active connection with configuration calls. Other drivers are request oriented. For these, the shadow restores the state of the driver and then resubmits to the driver any requests that were outstanding when the driver crashed.

As an example, for a failed sound card driver, the shadow driver resets the sound driver and all its open connections back to their pre-failures state. Specifically, the shadow scans its list of open connections and calls the open function in the driver to reopen each connection. The shadow then walks its log of configuration commands and replays any commands that set driver properties.

For some driver classes, the shadow cannot completely transfer its state into the driver. However, it may be possible to compensate in other, perhaps less elegant, ways. For example, a sound driver that is recording sound stores the number of bytes it has recorded since the last reset. After recovery, the sound driver initializes this counter to zero. Because no interface call is provided to change the counter value, the shadow driver must insert its “true” value into the return argument list whenever the application reads the counter to maintain the illusion that the driver has not crashed. The shadow can do this because it receives control (on its replicated call) before the kernel returns to user space.

After resetting driver and connection state, the shadow must handle requests that were either outstanding when the driver crashed or arrived while the driver was recovering. Unfortunately, shadow drivers cannot guarantee exactly-once behavior for driver requests and must rely on devices and higher levels of software to absorb duplicate requests. For example, if a driver crashes after submitting a request to a device but before notifying the kernel that the request has completed, the shadow cannot know whether the request was actually processed. During recovery, the shadow driver has two choices: restart in-progress requests and risk duplication, or cancel the request and risk lost data. For some device classes, such as disks or networks, duplication is acceptable. However, other classes, such as printers, may not tolerate duplicates. In these cases, the shadow driver cancels outstanding requests, which may limit its ability to mask failures.

After this final step, the driver has been reinitialized, linked into the kernel, reloaded with its pre-failure state, and is ready to process commands. At this point, the shadow driver notifies the shadow manager, which sets the taps to restore kernel-driver communication and reestablish passive-mode monitoring.

4.4 Active-Mode Proxying of Kernel Requests

While a shadow driver is restoring a failed driver, it is also acting in place of the driver to conceal the failure and recovery from applications and the kernel. The shadow driver's response to a driver request depends on the driver class and request semantics. In general, the shadow will take one of five actions: (1) respond with information that it has recorded, (2) silently drop the request, (3) queue the request for later processing, (4) block the request until the driver recovers, or (5) report that the driver is busy and the kernel or application should try again later. The choice of strategy depends on the caller's expectations of the driver.

Writing a shadow driver that proxies for a failed driver requires knowledge of the kernel-driver interface, interactions, and requirements. For example, the kernel may require that some driver functions never block, while others always block. Some kernel requests are idempotent (e.g., many `ioctl` commands), permitting duplicate requests to be dropped, while others return different results on every call (e.g., many `read` requests). The shadow for a driver class uses these requirements to select the response strategy.

Active proxying is simplified for driver interfaces that support a notion of "busy." By reporting that the device is currently busy, shadow drivers instruct the kernel or application to block calls to a driver. For example, network drivers in Linux may reject requests and turn themselves off if their queues are full. The kernel then refrains from sending packets until the driver turns itself back on. Our shadow network driver exploits this behavior during recovery by returning a "busy" error on calls to send packets. IDE storage drivers support a similar notion when request queues fill up. Sound drivers can report that their buffers are temporarily full.

Our shadow sound-card driver uses a mix of all five strategies for emulating functions in its service interface. The shadow blocks kernel `read` and `write` requests, which play or record sound samples, until the failed driver recovers. It processes `ioctl` calls itself, either by responding with information it captured or by logging the request to be processed later. For `ioctl` commands that are idempotent, the shadow driver silently drops duplicate requests. Finally, when applications query for buffer space, the shadow responds that buffers are full. As a result, many applications block themselves rather than blocking in the shadow driver.

4.5 Limitations

As previously described, shadow drivers have limitations. First, shadow drivers rely on dynamic unloading and reloading of device drivers. If a driver cannot be reloaded dynamically, or will not reinitialize properly, then a shadow cannot recover the driver. Second, shadow

drivers rely on explicit communication between the device driver and kernel. If driver-kernel communication takes place through an ad-hoc interface, such as shared memory, the shadow driver cannot monitor it. Third, shadow drivers assume that driver failure does not cause irreversible side effects. If a corrupted driver stores persistent state (e.g., printing a bad check or writing bad data on a disk), the shadow driver will not be able to correct that action.

The effectiveness of shadow drivers is also limited by the abilities of the isolation and failure-detection subsystem. If this layer cannot prevent kernel corruption, then shadow drivers cannot facilitate system recovery. In addition, if the fault-isolation subsystem does not detect a failure, then shadow drivers will not be properly invoked to perform recovery, and applications may fail. Detecting failures is difficult because drivers are complex and may respond to application requests in many ways. It may be impossible to detect a valid but incorrect return value; for example, a sound driver may return incorrect sound data when recording. As a result, no failure detector can detect every device driver failure. However, we support class-based failure detectors that can detect violations of a driver's programming interface and reduce the number of undetected failures.

Finally, shadow drivers may not be suitable for applications with real-time demands. During recovery, a device may be unavailable for several seconds without notifying the application of a failure. These applications, which should be written to tolerate failures, would be better served by a solution that restarts the driver but does not perform active proxying.

4.6 Summary

This section presented the details of our Linux shadow driver implementation. The shadow driver concept is straightforward: passively monitor normal operations, proxy during failure, and reintegrate during recovery. Ultimately, the value of shadow drivers depends on the degree to which they can be implemented correctly, efficiently, and easily in an operating system. The following section evaluates some of these questions both qualitatively and quantitatively.

5 Evaluation

This section evaluates four key aspects of shadow drivers.

1. *Performance.* What is the performance overhead of shadow drivers during normal, passive-mode operation (i.e., in the absence of failure)? This is the dynamic cost of our mechanism.

Class	Driver	Device
Network	e1000	Intel Pro/1000 Gigabit Ethernet
	pcnet32	AMD PCnet32 10/100 Ethernet
	3c59x	3COM 3c509b 10/100 Ethernet
	e100	Intel Pro/100 Ethernet
	epic100	SMC EtherPower 10/100 Ethernet
Sound	audigy	SoundBlaster Audigy sound card
	emu10k1	SoundBlaster Live! sound card
	sb	SoundBlaster 16 sound card
	es1371	Ensoniq sound card
	cs4232	Crystal sound card
	i810_audio	Intel 810 sound card
Storage	ide-disk	IDE disk
	ide-cd	IDE CD-ROM

Table 1: The three classes of shadow drivers and the Linux drivers tested. We present results for the boldfaced drivers only, as the others behaved similarly.

2. *Fault-Tolerance*. Can applications that use a device driver continue to run even after the driver fails? We evaluate shadow driver recovery in the presence of simple failures to show the benefits of shadow drivers compared to a system that provides failure isolation alone.
3. *Limitations*. How reasonable is our assumption that driver failures are fail-stop? Using synthetic fault injection, we evaluate how likely it is that driver failures are fail-stop.
4. *Code size*. How much code is required for shadow drivers and their supporting infrastructure? We evaluate the size and complexity of the shadow driver implementation to highlight the engineering cost integrating shadow drivers into an existing system.

Based on a set of controlled application and driver experiments, our results show that shadow drivers: (1) impose relatively little performance overhead, (2) keep applications running when a driver fails, (3) are limited by a system's ability to detect that a driver has failed, and (4) can be implemented with a modest amount of code.

The experiments were run on a 3 GHz Pentium 4 PC with 1 GB of RAM and an 80 GB, 7200 RPM IDE disk drive. We built and tested three Linux shadow drivers for three device-driver classes: network interface controller, sound card, and IDE storage device. To ensure that our generic shadow drivers worked consistently across device driver implementations, we tested them on thirteen different Linux drivers, shown in Table 1. Although we present detailed results for only one driver in each class (*e1000*, *audigy*, and *ide-disk*), behavior across all drivers was similar.

Device Driver	Application Activity
<i>Sound</i> (<i>audigy driver</i>)	<ul style="list-style-type: none"> • mp3 player (<i>zinf</i>) playing 128kb/s audio • audio recorder (<i>audacity</i>) recording from microphone • speech synthesizer (<i>festival</i>) reading a text file • strategy game (<i>Battle of Wesnoth</i>)
<i>Network</i> (<i>e1000 driver</i>)	<ul style="list-style-type: none"> • network send (<i>netperf</i>) over TCP/IP • network receive (<i>netperf</i>) over TCP/IP • network file transfer (<i>scp</i>) of a 1GB file • remote window manager (<i>vnc</i>) • network analyzer (<i>ethereal</i>) sniffing packets
<i>Storage</i> (<i>ide-disk driver</i>)	<ul style="list-style-type: none"> • compiler (<i>make/gcc</i>) compiling 788 C files • encoder (<i>LAME</i>) converting 90 MB file .wav to .mp3 • database (<i>mysql</i>) processing the <i>Wisconsin Benchmark</i>

Table 2: The applications used for evaluating shadow drivers.

5.1 Performance

To evaluate performance, we produced three OS configurations based on the Linux 2.4.18 kernel:

1. *Linux-Native* is the unmodified Linux kernel.
2. *Linux-Nooks* is a version of *Linux-Native* that includes the Nooks fault isolation subsystem but no shadow drivers. When a driver fails, this system restarts the driver but does not attempt to conceal its failure.
3. *Linux-SD* is a version of *Linux-Nooks* that includes our entire recovery subsystem, including the Nooks fault isolation subsystem, the shadow manager, and our three shadow drivers.

We selected a variety of common applications that depend on our three device driver classes and measured their performance. The application names and behaviors are shown in Table 2.

Different applications have different performance metrics of interest. For the disk and sound drivers, we ran the applications shown in Table 2 and measured elapsed time. For the network driver, throughput is a more useful metric; therefore, we ran the throughput-oriented *network send* and *network receive* benchmarks. For all drivers, we also measured CPU utilization while the programs ran. All measurements were repeated several times and showed a variation of less than one percent.

Figure 5 shows the performance of *Linux-Nooks* and *Linux-SD* relative to *Linux-Native*. Figure 6 compares CPU utilization for execution of the same applications on the three OS versions. Both figures make clear that shadow drivers impose only a small performance penalty compared to running with no isolation at all, and no additional penalty beyond that imposed by isolation

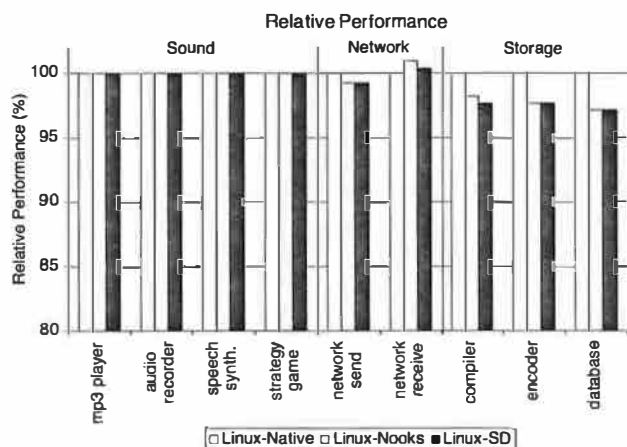


Figure 5: Comparative application performance, relative to *Linux-Native*, for three configurations. The X-axis crosses at 80%.

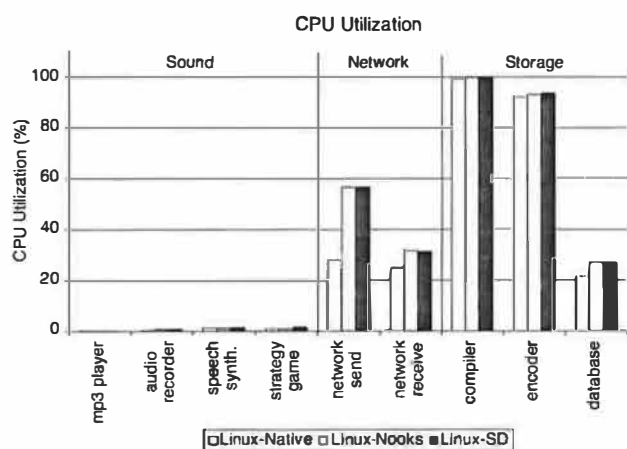


Figure 6: Absolute CPU utilization by application for three configurations.

alone. Across all nine applications, performance of the system with shadow drivers averaged 99% of the system without, and was never worse than 97%.

The low overhead of shadow drivers can be explained in terms of its two constituents: fault isolation and the shadowing itself. As mentioned previously, fault isolation runs each driver in its own domain, leading to overhead caused by domain crossings. Each domain crossing takes approximately 3000 cycles, mostly to change page tables and execution stacks. As a side effect of changing page tables, the Pentium 4 processor flushes the TLB, resulting in TLB misses that can noticeably slow down drivers [33].

For example, the kernel calls the driver approximately 1000 times per second when running *audio recorder*. Each invocation executes only a small amount of code. As a result, isolating the sound driver adds only negligibly to CPU utilization, because there are not many cross-

ings and not much code to slow down. For the most disk-intensive of the IDE storage applications, the *database* benchmark, the kernel and driver interact only 290 times per second. However, each call into the *ide-disk* driver results in substantial work to process a queue of disk requests. The TLB-induced slowdown doubles the time *database* spent in the driver relative to *Linux-Native* and increases the application's CPU utilization from 21% to 27%. On the other hand, the *network send* benchmark transmits 45,000 packets per second, causing 45,000 domain crossings. The driver does little work for each packet, but the overall impact is visible in Figure 6, where CPU utilization for this benchmark increases from 28% to 57% with driver fault isolation.

In the case the actual shadowing, we see from a comparison of the *Linux-Nooks* and *Linux-SD* bars in Figures 5 and 6 that the cost is small or negligible. As noted in Section 4.2, many passive-mode shadow-driver functions are no-ops. As a result, the incremental passive-mode performance cost over basic fault isolation is low or unmeasurable in many cases.

In summary, then, the overall performance penalty of shadow drivers during failure-free operation is low, suggesting that shadow drivers could be used across a wide range of applications and environments.

5.2 Fault-Tolerance

Regardless of performance, the crucial question for shadow drivers is whether an application can continue functioning following the failure of a device driver on which it relies. To answer this question, we tested 10 applications on the three configurations, *Linux-Native*, *Linux-Nooks*, and *Linux-SD*. For the disk and sound drivers, we again ran the applications shown in Table 2. Because we were interested in the response to, not performance, we substituted *network file copy*, *remote window manager*, and *network analyzer* for the networking benchmarks.

We simulated common bugs by injecting a software fault into a device driver while an application using that driver was running. Because both *Linux-Nooks* and *Linux-SD* depend on the same isolation and failure-detection services, we differentiate their recovery capabilities by simulating failures that are easily isolated and detected. To generate realistic synthetic driver bugs, we analyzed patches posted to the Linux Kernel Mailing List [24]. We found 31 patches that contained the strings “patch,” “driver,” and “crash” or “oops” (the Linux term for a kernel fault) in their subject lines. Of the 31 patches, we identified 11 that fix transient bugs (i.e., bugs that occur occasionally or only after a long delay from the triggering test). The most common cause of failure (three instances) was a missing check for a null pointer, often with a secondary cause of missing or broken synchronization.

Device Driver	Application Activity	Application Behavior		
		Linux-Native	Linux-Nooks	Linux-SD
<i>Sound</i> (<i>audigy driver</i>)	mp3 player	CRASH	MALFUNCTION	✓
	audio recorder	CRASH	MALFUNCTION	✓
	speech synthesizer	CRASH	✓	✓
	strategy game	CRASH	MALFUNCTION	✓
<i>Network</i> (<i>e1000 driver</i>)	network file transfer	CRASH	✓	✓
	remote window manager	CRASH	✓	✓
	network analyzer	CRASH	MALFUNCTION	✓
<i>IDE</i> (<i>ide-disk driver</i>)	compiler	CRASH	CRASH	✓
	encoder	CRASH	CRASH	✓
	database	CRASH	CRASH	✓

Table 3: The observed behavior of several applications following the failure of the device drivers on which they rely. There are three behaviors: a checkmark (✓) indicates that the application continued to operate normally; CRASH indicates that the application failed completely (i.e., it terminated); MALFUNCTION indicates that the application continued to run, but with abnormal behavior.

We also found missing pointer initialization code (two instances) and bad calculations (two instances) that led to endless loops and buffer overruns. Because these faults are detected by Nooks, they cause fail-stop failures on *Linux-Nooks* and *Linux-SD*.

We injected a null-pointer dereference bug derived from these patches into our three drivers. We ensured that the synthetic bug was transient by inserting the bug into uncommon execution paths, such as code that handles unusual hardware conditions. These paths are rarely executed, so we accelerated the occurrence of faults by also executing the bug at random intervals. The fault code remains active in the driver during and after recovery.

Table 3 shows the three application behaviors we observed. When a driver failed, each application either continued to run normally (✓), failed completely (“CRASH”), or continued to run but behaved abnormally (“MALFUNCTION”). In the latter case, manual intervention was typically required to reset or terminate the program.

This table demonstrates that shadow drivers (*Linux-SD*) enable applications to continue running normally even when device drivers fail. In contrast, all applications on *Linux-Native* failed when drivers failed. Most programs running on *Linux-Nooks* failed or behaved abnormally, illustrating that Nooks’ kernel-focused recovery system does not extend to applications. For example, Nooks isolates the kernel from driver faults and reboots (unloads, reloads, and restarts) the driver. However, it lacks two key features of shadow drivers: (1) it does not advance the driver to its pre-fail state, and (2) it has no component to “pinch hit” for the failed driver during recovery. As a result, *Linux-Nooks* handles driver failures by returning an error to the application, leaving it to recover by itself. Unfortunately, few applications can do this.

Some applications on *Linux-Nooks* survived the driver failure but in a degraded form. For example, *mp3 player*, *audio recorder* and *strategy game* continued running, but they lost their ability to input or output sound until the user intervened. Similarly, *network analyzer*, which interfaces directly with the network device driver, lost its ability to receive packets once the driver was reloaded.

A few applications continued to function properly after driver failure on *Linux-Nooks*. One application, *speech synthesizer*, includes the code to reestablish its context within an unreliable sound card driver. Two of the network applications survived on *Linux-Nooks* because they access the network device driver through kernel services (TCP/IP and sockets) that are themselves resilient to driver failures.

Linux-SD recovers transparently from disk driver failures. Recovery is possible because the IDE storage shadow driver instance maintains the failing driver’s initial state. During recovery the shadow copies back the initial data and reuses the driver code, which is already stored read-only in the kernel. In contrast, *Linux-Nooks* illustrates the risk of circular dependencies from rebooting drivers. Following these failures, Nooks, which had unloaded the *ide-disk* driver, was then required to reload the driver off the IDE disk. The circularity could only be resolved by a system reboot. While a second (non-IDE) disk would mitigate this problem, few machines are configured this way.

In general, programs that directly depend on driver state but are unprepared to deal with its loss benefit the most from shadow drivers. In contrast, those that do not directly depend on driver state or are able to reconstruct it when necessary benefit the least. Our experience suggests that few applications are as fault-tolerant as *speech synthesizer*. Were future applications to be pushed in this direction, software manufacturers would either need to

develop custom recovery solutions on a per-application basis or find a general solution that could protect any application from the failure of a kernel device driver. Cost is a barrier to the first approach. Shadow drivers are a path to the second.

Application Behavior During Driver Recovery

Although shadow drivers can prevent application failure, they are not “real” device drivers and do not provide complete device services. As a result, we often observed a slight timing disruption while the driver recovered. At best, output was queued in the shadow driver. At worst, input was lost by the device. The length of the delay was primarily determined by the recovering device driver itself, which, on initialization, must first discover, and then configure, the hardware.

Few device drivers implement fast reconfiguration, which can lead to brief recovery delays. For example, the temporary loss of the *e1000* network device driver prevented applications from receiving packets for about five seconds.² Programs using files stored on the disk managed by the *ide-disk* driver stalled for about four seconds during recovery. In contrast, the normally smooth sounds produced by the *audigy* sound card driver were interrupted by a pause of about one-tenth of one second, which sounded like a slight click in the audio stream.

Of course, the significance of these delays depends on the application. Streaming applications may become unacceptably “jittery” during recovery. Those processing input data in real-time might become lossy. Others may simply run a few seconds longer in response to a disk that appears to be operating more sluggishly than usual. In any event, a short delay during recovery is best considered in light of the alternative: application and system failure.

5.3 Limits to Recovery

The previous section assumed that failures were fail-stop. However, driver failures experienced in deployed systems may exhibit a wider variety of behaviors. For example, a driver may corrupt state in the application, kernel, or device without the failure being detected. In this situation, shadow drivers may not be able to recover or mask failures from applications. This section uses fault injection experiments in an attempt to generate faults that may not be fail-stop.

Non-fail-stop Failures

If driver failures are not fail stop, then shadow drivers may not be useful. To evaluate whether device driver fail-

²This driver is particularly slow at recovery. The other network drivers we tested recovered in less than a second.

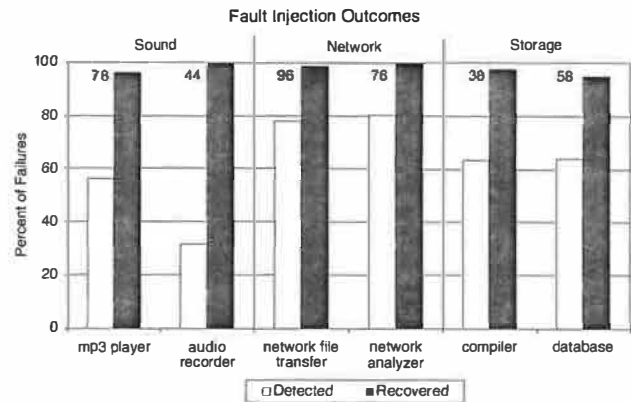


Figure 7: Results of fault-injection experiments on Linux-SD. We show (1) the percentage of failures that are automatically detected by the fault isolation subsystem, and (2) the percentage of failures that shadow drivers successfully recovered. The total number of failures experienced by each application is shown at the top of the chart.

ures are indeed fail-stop, we performed large-scale fault-injection tests of our drivers and applications running on Linux-SD. For each driver and application combination, we ran 350 fault-injection trials.³ In total, we ran 2100 trials across the three drivers and six applications. Between trials, we reset the system and reloaded the driver. For each trial, we injected five random errors into the driver while the application was using it. We ensured the errors were transient by removing them during recovery. After injection, we visually observed the impact on the application and the system to determine whether a failure or recovery had occurred. For each driver, we tested two applications with significantly different usage scenarios. For example, we chose one sound-playing application (*mp3 player*) and one sound-recording application (*audio recorder*).

If we observed a failure, we then assessed the trial on two criteria: whether the fault was detected, and whether the shadow driver could mask the failure and subsequent recovery from the application. For undetected failures, we triggered recovery manually. Note that a user may observe a failure that an application does not, for example, by testing the application’s responsiveness.

Figure 7 shows the results of our experiments. For each application, we show the percentage of failures that the Nooks subsystem detected and the percentage of failures from which shadow drivers correctly recovered. Only 18% of the injected faults caused a visible failure.

In our tests, 390 failures occurred across all applications. The system automatically detected 65% of the failures. In every one of these cases, shadow drivers were able to mask the failure and facilitate driver recovery. The

³For details on the fault injector see [33].

Driver Class	Shadow Driver Lines of Code	Device Driver Shadowed Lines of Code	Class Size # of Drivers	Class Size Lines of Code
Sound	666	7,381 (<i>audigy</i>)	48	118,981
Network	198	13,577 (<i>e1000</i>)	190	264,500
Storage	321	5,358 (<i>ide-disk</i>)	8	29,000

Table 4: Size and quantity of shadows and the drivers they shadow.

system failed to detect 35% of the failures. In these cases, we manually triggered recovery. Shadow drivers recovered from nearly all of these failures (127 out of 135). Recovery was unsuccessful in the remaining 8 cases because either the system had crashed (5 cases) or the driver had corrupted the application beyond the possibility of recovery (3 cases). It is possible that recovery would have succeeded had these failures been detected earlier with a better failure detector.

Across all applications and drivers, we found three major causes of undetected failure. First, the system did not detect application hangs caused by I/O requests that never completed. Second, the system did not detect errors in the interactions between the device and the driver, e.g., incorrectly copying sound data to a sound card. Third, the system did not detect certain bad parameters, such as incorrect result codes or data values. Detecting these three error conditions would require that the system better understand the semantics of each driver class. For example, 68% of the sound driver failures with *audio recorder* went undetected. This application receives data from the driver in real time and is highly sensitive to driver output. A small error or delay in the results of a driver request may cause the application to stop recording or record the same sample repeatedly.

Our results demonstrate a need for class-based failure detectors that can detect violations of the driver interface to achieve high levels of reliability. However, driver failures need not be detected quickly to be fail-stop. There was a significant delay between the failure and the subsequent manual recovery in our tests, and yet the applications survived the vast majority of undetected failures. Thus, even a slow failure detector can be effective at improving application reliability.

Non-transient Failures

Shadow drivers can recover from transient failures only. In contrast, deterministic failures may recur during recovery when the shadow configures the driver. While unable to recover, shadow drivers are still useful for these failures. When a failure recurs during recovery, the sequence of shadow driver recovery events creates a detailed reproduction scenario that aids diagnosis. This record of recovery contains the driver's calls into the kernel, re-

quests to configure the driver, and I/O requests that were pending at the time of failure. This information enables a software engineer to find and fix the offending bug more efficiently.

5.4 Code Size

The preceding sections evaluated the *efficiency* and *effectiveness* of shadow drivers. This section examines the *complexity* of shadow drivers in terms of code size, which can serve as a proxy for complexity.

Table 4 shows, for each class, the size in lines of code of the shadow driver for the class. For comparison, we show the size of the driver from the class that we tested and the total number and cumulative size of existing Linux device drivers in that class in the 2.4.18 kernel. The total code size is an indication of the leverage gained through the shadow's class-driver structure. Furthermore, the table shows that a shadow driver is significantly smaller than the device driver it shadows. For example, our sound-card shadow driver is only 9% of the size of the *audigy* device driver it shadows. The IDE storage shadow is only 6% percent of the size of the Linux *ide-disk* device driver.

The Nooks driver fault isolation subsystem we built upon contains about 23,000 lines of code. In total, we added about 3300 lines of new code to Nooks to support our three class drivers. Otherwise, we made no changes to the remainder of the Linux kernel. Shadow drivers required the addition of approximately 600 lines of code for the shadow manager, 800 lines of common code shared by all shadow drivers, and another 750 lines of code for general utilities. Of the 177 taps we inserted, only 31 required actual code; the remainder were no-ops.

5.5 Summary

This section examined the performance, fault-tolerance, limits, and code size of shadow drivers. Our results demonstrate that: (1) the performance overhead of shadow drivers during normal operation is small, particularly when compared to a purely isolating system, (2) applications that failed in any form on *Linux-Native* or *Linux-Nooks* ran normally with shadow drivers, (3) the reliability provided by shadow drivers is limited by the system's ability to detect failures, and (4) shadow drivers

are small, even relative to single device driver. Overall, these results indicate that shadow drivers have the potential to significantly improve the reliability of applications on modern operating systems with only modest cost.

6 Conclusions

Improving the reliability of modern systems demands that we increase their resilience. To this end, we designed and implemented *shadow drivers*, which mask device driver failures from both the operating system and applications.

Our experience shows that shadow drivers improve application reliability, by concealing a driver's failure while facilitating recovery. A single shadow driver can enable recovery for an entire class of device drivers. Shadow drivers are also efficient, imposing little performance degradation. Finally, they are transparent, requiring no code changes to existing drivers.

Acknowledgments

This work was supported in part by the National Science Foundation under grants ITR-0085670 and CCR-0121341. We would also like to thank our shepherd, Peter Chen, who provided many valuable insights.

References

- [1] S. Arthur. Fault resilient drivers for Longhorn server. Technical Report WinHec 2004 Presentation DW04012, Microsoft Corporation, May 2004.
- [2] Ö. Babaoğlu. Fault-tolerant computing based on Mach. In *Proceedings of the USENIX Mach Symposium*, Oct. 1990.
- [3] R. Barga, D. Lomet, and G. Weikum. Recovery guarantees for general multi-tier applications. In *International Conference on Data Engineering*, 2002. IEEE.
- [4] J. F. Bartlett. A NonStop kernel. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, Dec. 1981.
- [5] A. Borg, W. Balu, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, Feb. 1989.
- [6] D. P. Bovet and M. Cesati. *Inside the Linux Kernel*. O'Reilly & Associates, 2002.
- [7] T. C. Bressoud. TFT: A software system for application-transparent fault tolerance. In *Proceedings of the 28th Symposium on Fault-Tolerant Computing*, June 1998. IEEE.
- [8] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, Feb. 1996.
- [9] G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proceedings of the Eighth IEEE Workshop on Hot Topics in Operating Systems*, May 2001.
- [10] S. Chandra and P. M. Chen. How fail-stop are faulty programs? In *Proceedings of the 28th Symposium on Fault-Tolerant Computing*, June 1998. IEEE.
- [11] S. Chandra and P. M. Chen. Whither generic recovery from application faults? A fault study using open-source software. In *Proceedings of the 2000 IEEE International Conference on Dependable Systems and Networks*, June 2000.
- [12] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio file cache: Surviving operating system crashes. In *Proceedings of the Seventh ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.
- [13] T. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, Dec. 1999.
- [14] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Oct. 2001.
- [15] D. R. Engler, M. F. Kaashoek, and J. O. Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Dec. 1995.
- [16] W. Feng. Making a case for efficient supercomputing. *ACM Queue*, 1(7), Oct. 2003.
- [17] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: a substrate for OS language and research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Oct. 1997.
- [18] J. Gray. Why do computers stop and what can be done about it? Technical Report 85-7, Tandem Computers, June 1985.
- [19] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
- [20] S. M. Hand. Self-paging in the Nemesis operating system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, Feb. 1999.
- [21] D. Jewett. Integrity S2: A fault-tolerant Unix platform. In *Proceedings of the 21st Symposium on Fault-Tolerant Computing*, June 1991. IEEE.
- [22] M. J. Kilgard, D. Blythe, and D. Hohn. System support for OpenGL direct rendering. In *Proceedings of Graphics Interface*, May 1995. Canadian Human-Computer Communications Society.
- [23] J. Liedtke. On μ -kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Dec. 1995.
- [24] Linux Kernel Mailing List. Available at <http://www.uwsg.indiana.edu/hypermail/linux/kernel>.
- [25] D. E. Lowell, S. Chandra, and P. M. Chen. Exploring failure transparency and the limits of generic recovery. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2000.
- [26] D. E. Lowell and P. M. Chen. Discount checking: Transparent, low-overhead recovery for general applications. Technical Report CSE-TR-410-99, University of Michigan, Nov. 1998.

- [27] G. Muller, M. Banâtre, N. Peyrouze, and B. Rochat. Lessons from FTM: An experiment in design and implementation of a low-cost fault-tolerant system. *IEEE Transactions on Software Engineering*, 45(2):332–339, June 1996.
- [28] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kȳcȳman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery-Oriented Computing (ROC): Motivation, definition, techniques, and case studies. Technical Report CSD-02-1175, UC Berkeley Computer Science, Mar. 2002.
- [29] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In *Proceedings of the 1995 Winter USENIX Conference*, Jan. 1995.
- [30] R. Short, Vice President of Windows Core Technology, Microsoft Corp. private communication, 2003.
- [31] M. Russinovich, Z. Segall, and D. Siewiorek. Application transparent fault management in Fault Tolerant Mach. In *Proceedings of the 23rd Symposium on Fault-Tolerant Computing*, June 1993. IEEE.
- [32] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, Oct. 1996.
- [33] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 22(4), Nov. 2004.
- [34] V. Orgovan, Systems Crash Analyst, Windows Core OS Group, Microsoft Corp. private communication, 2004.
- [35] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, Dec. 1993.
- [36] R. S. Wahbe and S. E. Lucco. Methods for safe and efficient implementation of virtual machines, June 1998. US Patent 5,761,477.
- [37] J. A. Whittaker. Software’s invisible users. *IEEE Software*, 18(3):84–88, May 2001.
- [38] W. A. Wulf. Reliable hardware-software architecture. In *Proceedings of the International Conference on Reliable Software*, 1975.
- [39] M. Young, M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, and A. Tevanian. Mach: A new kernel foundation for UNIX development. In *Proceedings of the 1986 Summer USENIX Conference*, June 1986.

Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines

Joshua LeVasseur

Volkmar Uhlig

Jan Stoess

Stefan Götz

University of Karlsruhe, Germany

Abstract

We propose a method to reuse unmodified device drivers and to improve system dependability using virtual machines. We run the unmodified device driver, with its original operating system, in a virtual machine. This approach enables extensive reuse of existing and unmodified drivers, independent of the OS or device vendor, significantly reducing the barrier to building new OS endeavors. By allowing distinct device drivers to reside in separate virtual machines, this technique isolates faults caused by defective or malicious drivers, thus improving a system's dependability.

We show that our technique requires minimal support infrastructure and provides strong fault isolation. Our prototype's network performance is within 3–8% of a native Linux system. Each additional virtual machine increases the CPU utilization by about 0.12%. We have successfully reused a wide variety of unmodified Linux network, disk, and PCI device drivers.

1 Introduction

The majority of today's operating system code base is accounted for by device drivers.¹ This has two major implications. First, any OS project that aims for even a reasonable breadth of device drivers faces either a major development and testing effort or has to support and integrate device drivers from a driver-rich OS (e.g., Linux or Windows). Even though almost all research OS projects reuse device drivers to a certain extent, full reuse for a significant driver base has remained an elusive goal and so far can be considered unachieved. The availability of drivers solely in binary format from the Windows driver base shows the limitations of integration and wrapping approaches as advocated by the OS-Kit project [10]. Also, implicit, undocumented, or in the worst case incorrectly documented OS behavior makes driver reuse with a fully emulated execution environment questionable.

The second implication of the large fraction of driver code in mature OS's is the extent of programming errors [7]. This is particularly problematic since testing requires accessibility to sometimes exotic or outdated

hardware. The likelihood of programming errors in commonly used device drivers is probably much lower than in application code; however, such errors are often fatal. Device drivers, traditionally executing in privileged mode, can potentially propagate faults to other parts of the operating system, leading to sporadic system crashes.

In this paper we propose a pragmatic approach for full reuse and strong isolation of legacy device drivers. Instead of integrating device driver code we leave all drivers in their original and fully compatible execution environment—the original operating system. We run the device driver wrapped in the original operating system in a dedicated virtual machine (VM). Thus we can (almost) guarantee that semantics are preserved and that incompatibilities are limited to timing behavior introduced by virtual machine multiplexing.

The virtual machine environment also strongly isolates device drivers from the rest of the system to achieve fault containment. The isolation granularity depends on the number of collocated drivers in a single VM. By instantiating multiple collaborating VMs we can efficiently isolate device drivers with minimal resource overhead.

Reuse of device drivers and driver isolation are two important aspects of operating systems; however, they are usually discussed independently. With virtual machines, we propose to use a single abstraction to solve both problems in an extremely flexible, elegant, and efficient way.

2 Related Work

Our work uses known principles of hardware-based isolation to achieve driver reuse and improved system dependability. It is unique in the manner and the extent to which it accomplishes unmodified driver reuse, and how it improves system dependability, in terms of drivers, without system modification.

2.1 Reuse

Binary driver reuse has been achieved with cohosting, as used in VMware Workstation [32]. Cohosting multiplexes the processor between two collaborating operating systems, e.g., the driver OS and the VM monitor.

¹Linux 2.4.1 drivers cover 70% of its IA32 code base [7].

When device activity is necessary, processor control is transferred to the driver OS in a world switch (which restores the interrupt handlers of the driver OS, etc.). The driver OS releases ownership of the processor upon device activity completion. The cohosting method offers no trust guarantees; both operating systems run fully privileged in supervisor mode and can interfere with each other.

Device drivers are commonly reused by transplanting source modules from a donor OS into the new OS [2, 4, 11, 15, 28, 35]. In contrast to cohosting, the new OS dominates the transplanted drivers. The transplant merges two independently developed code bases, glued together with support infrastructure. Ideally the two subsystems enjoy independence, such that the design of one does not interfere with the design of the other. Past work demonstrates that, despite great effort, conflicts are unavoidable and lead to compromises in the structure of the new OS. Transplantation has several categories of reuse issues, which we further describe.

Semantic Resource Conflicts

The transplanted driver obtains resources (memory, locks, CPU, etc.) from its new OS, subject to normal obligations and limitations, creating a new and risky relationship between the two components. In the reused driver's raw state, its manner of resource use could violate the resource's constraints. The misuse can cause accidental denial of service (e.g., the reused driver's non-preemptible interrupt handler consumes enough CPU to reduce the response latency for other subsystems), can cause corruption of a manager's state machine (e.g., invoking a non-reentrant memory allocator at interrupt time [15]), or can dead-lock in a multiprocessor system.

These semantic conflicts are due to the nature of OS design. A traditional OS divides bulk platform resources such as memory, processor time, and interrupts between an assortment of subsystems. The OS refines the bulk resources into linked lists, timers, hash tables, top-halves and bottom-halves, and other units acceptable for distributing and multiplexing between the subsystems. The resource refinements impose rules on the use of the resources, and depend on cooperation in maintaining the integrity of the state machines. Modules of independent origin substitute a glue layer for the cooperative design. For example, when a Linux driver waits for I/O, it removes the current thread from the run queue. To capture the intended thread operation and to map it into an operation appropriate for the new OS, the glue layer allocates a Linux thread control block when entering a reused Linux component [2, 28]. In systems that use asynchronous I/O, the glue layer converts the thread operations into I/O continuation objects [15].

Sharing Conflicts

A transplanted driver shares the address space and privilege domain with the new OS. Their independently developed structures contend for the same resources in these two domains, and are subject to each other's faults.

Due to picky device drivers and non-modular code, a solution for fair address space sharing may be unachievable. The older Linux device drivers, dedicated to the IA32 platform, assumed virtual memory was idempotently mapped to physical memory. Reuse of these drivers requires modifications to the drivers or loss in flexibility of the address space layout. The authors in [28] decided not to support such device drivers, because the costs conflicted with their goals. The authors of [15] opted to support the drivers by remapping their OS.

Privileged operations generally have global side effects. When a device driver executes a privileged operation for the purposes of its local module, it likely affects the entire system. A device driver that disables processor interrupts disables them for all devices. Cooperatively designed components plan for the problem; driver reuse spoils cooperative design.

Engineering Effort

Device driver reuse reduces engineering effort in OS construction by avoiding reimplementation of the device drivers. Preserving confidence in the correctness of the original drivers is also important. When given device drivers that are already considered to be reliable and correct (error counts tend to reduce over time [7]), it is hoped that their reuse will carry along the same properties. Confidence in the new system follows from thorough knowledge of the principles behind the system's construction, accompanied by testing.

Reusing device drivers through transplantation reduces the overall engineering effort for constructing a new OS, but it still involves substantial work. In [10] Ford et al. report 12% of the OS-Kit code as glue code.

Engineering effort is necessary to extract the reused device drivers from their source operating systems, and to compile and link with the new operating system. The transplant requires glue layers to handle semantic differences and interface translation.

For implementation of a glue layer that gives us confidence in its reliability, intimate knowledge is required about the functionality, interfaces, and semantics of the reused device drivers. The authors in [2, 15, 28] all demonstrate intimate knowledge of their source operating systems.

The problems of semantic and resource conflicts multiply as device drivers from several source operating systems are transplanted into the new OS. Intimate knowledge of the internals of each source operating system

is indispensable. Driver update tracking can necessitate adaptation effort as well.

2.2 Dependability

The use of virtual machines to enhance reliability has been long known [16]. A variety of other techniques for enhancing system dependability also exist, such as safe languages and software isolation, and are complementary to our approach. The orthogonal design provided by our solution permits coexistence with incompatible subsystems and development methodologies.

User-level device driver frameworks [9, 11, 17, 20, 26, 31] are a known technique to improve dependability. They are typically deployed in a microkernel environment. Our approach also executes the device drivers at user level; however, we use the platform interface rather than a specialized and potentially more efficient API.

The recent Nooks project [33] shares our goal of retrofitting dependability enhancements in commodity systems. Their solution isolates drivers within protection domains, yet still executes them within the kernel with complete privileges. Without privilege isolation, complete fault isolation is not achieved, nor is detection of malicious drivers possible.

Nooks collocates with the target kernel, adding 22,000 lines of code to the Linux kernel's large footprint, all privileged. The Nooks approach is similar to second generation microkernels (such as L4, EROS, or K42) in providing address space services and synchronous communication across protection domains, but it doesn't take the next step to deprive the isolation domains (and thus exit to user-level, which is a minuscule overhead compared to the cost of address space switching on IA32).

To compensate for Linux's intricate subsystem entanglement, Nooks includes interposition services to maintain the integrity of resources shared between drivers. In our approach, we connect drivers at a high abstraction level—the request—and thus avoid the possibility of corrupting one driver by the actions of another driver.

Like us, another contemporary project [12, 13] uses paravirtualization for user-level device drivers, but focuses on achieving a unified device API and driver isolation. Our approach specifically leaves driver interfaces undefined and thus open for specializations and layer-cutting optimizations. Their work argues for a set of universal common-denominator interfaces per device class.

3 Approach

The traditional approach to device driver construction favors intimate relationships between the drivers and their kernel environments, interfering with easy reuse of

drivers. On the other hand, *applications* in the same environments interface with their kernels via well defined APIs, permitting redeployment on similar kernels. Applications enjoy the benefits of orthogonal design.

To achieve reuse of device drivers from a wide selection of operating systems, we classify drivers as applications subject to orthogonal design, based on the following principles:

Resource delegation: The driver receives only bulk resources, such as memory at page granularity. The responsibility to further refine the bulk resources lies on the device driver. The device driver converts its memory into linked lists and hash tables, it manages its stack layout to support reentrant interrupts, and divides its CPU time between its threads.

Separation of name spaces: The device driver executes within its own address space. This requirement avoids naming conflicts between driver instances, and helps prevent faulty accesses to other memory.

Separation of privilege: Like applications, the device driver executes in unprivileged mode. It is unable to interfere with other OS components via privileged instructions.

Secure isolation: The device driver lacks access to the memory of non-trusting components. Likewise, the device driver is unable to affect the flow of execution in non-trusting components. These same properties also protect the device driver from the other system components. When non-trusting components share memory with the drivers, they are expected to protect their internal integrity; sensitive information is not stored on shared pages, or when it is, shadow copies are maintained in protected areas of the clients [14].

Common API: The driver allocates resources and controls devices with an API common to all device drivers. The API is well documented, well understood, powerfully expressive, and relatively static.

Most legacy device drivers in their native state violate these orthogonal design principles. They use internal interfaces of their native operating systems, expect refined resources, execute privileged instructions, and share a global address space. Their native operating systems partially satisfy our requirements. Operating systems provide resource delegation and refinement, and use a common API—the system's instruction set and platform architecture. By running the OS with the device driver in a virtual machine, we satisfy all of the principles and thus achieve orthogonal design.

3.1 Architecture

To reuse and isolate a device driver, we execute it and its native OS within a virtual machine. The driver directly controls its device via a pass-through enhancement to the virtual machine, which permits the device driver OS (DD/OS) to access the device's registers, ports, and receive hardware interrupts. The VM, however, inhibits the DD/OS from seeing and accessing devices which belong to other VMs.

The driver is reused by a client, which is any process in the system external to the VM, at a privileged or user level. The client interfaces with the driver via a translation module added to the device driver's OS. This module behaves as a server in a client-server model. It maps client requests into sequences of DD/OS primitives for accessing the device, and converts completed requests into appropriate responses to the client.

The translation module controls the DD/OS at one of several layers of abstraction: potentially the user-level API of the DD/OS (e.g., file access to emulate a raw disk), raw device access from user level (e.g., raw sockets), abstracted kernel module interfaces such as the buffer cache, or the kernel primitives of the device drivers in the DD/OS. It is important to choose the correct abstraction layer to achieve the full advantages of our device driver reuse approach; it enables a single translation module to reuse a wide variety of devices, hopefully without a serious performance penalty. For example, a translation module that interfaces with the block layer can reuse hard disks, floppy disks, optical media, etc., as opposed to reusing only a single device driver.

To isolate device drivers from each other, we execute the drivers in separate and co-existing virtual machines. This also enables simultaneous reuse of drivers from incompatible operating systems. When an isolated driver relies on another (e.g., a device needs bus services), then the two DD/OS's are assembled into a client-server relationship. See Figure 1 for a diagram of the architecture.

The requirement for a complete virtual machine implementation is avoidable by substituting a paravirtualized DD/OS for the unmodified DD/OS. In the paravirtualized model [3,16], the device driver's OS is modified to interface directly with the underlying system. However, most importantly, the device drivers in general remain unmodified; they only need to be recompiled.

3.2 Virtual Machine Environment

In our virtualization architecture we differentiate between five entities:

- The hypervisor is the privileged kernel, which securely multiplexes the processor between the virtual

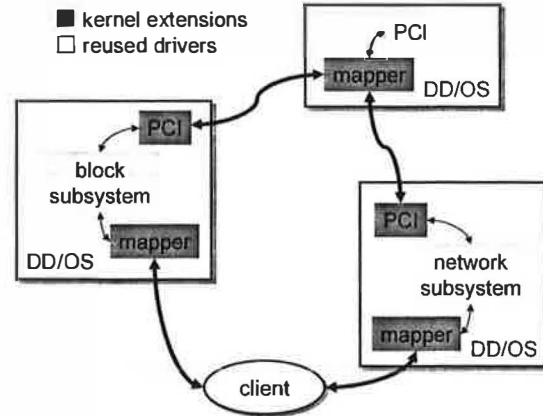


Figure 1: Device driver reuse and isolation. The kernel extensions represent the components loaded into the DD/OS's to coordinate device driver reuse. The block and network DD/OS's recursively use the PCI DD/OS.

machines. It runs in privileged mode and enforces protection for memory and IO ports.

- The virtual machine monitor (VMM) allocates and manages resources and implements the virtualization layer, such as translating access faults into device emulations. The VMM can be either collocated with the hypervisor in privileged mode or unprivileged and interacting with the hypervisor through a specialized interface.
- Device driver OS's host unmodified legacy device drivers and have pass-through access to the device. They control the device via either port IO or memory mapped IO and can initiate DMA. However, the VMM restricts access to only those devices that are managed by each particular DD/OS.
- Clients use device services exported by the DD/OS's, in a traditional client-server scenario. Recursive usage of driver OS's is possible; i.e. a client can act as a DD/OS for another client. The client could be the hypervisor itself.
- Translation modules are added to DD/OS's to provide device services to the clients. They provide the interface for the client-to-DD/OS communication, and map client requests into DD/OS primitives.

The hypervisor features a low-overhead communication mechanism for inter-virtual-machine communication. For message notification, each VM can raise a communication interrupt in another VM and thereby signal a pending request. Similarly, on request completion the DD/OS can raise a completion interrupt in the client OS.

The hypervisor provides a mechanism to share memory between multiple virtual machines. The VMM can

register memory areas of one VM in another VM's physical memory space, similarly to memory-mapped device drivers.

3.3 Client Requests

To provide access to its devices, the driver OS exports a virtual device interface that can be accessed by the client. The interface for client-to-DD/OS device communication is not defined by the hypervisor or the VMM but rather left to the specific translation module implementation. This allows for optimizations such as virtual interrupt coalescing, scatter-gather copying, shared buffers, and producer-consumer rings as used in Xen [3].

The translation module makes one or more memory pages accessible to the client OS and uses interrupts for signalling, subject to the particular interface and request requirements. This is very similar to interaction with real hardware devices. When the client signals the DD/OS, the VMM injects a virtual interrupt to cause invocation of the translation module. When the translation module signals the client in response, it invokes a method of the VMM, which can be implemented as a trap due to a specific privileged instruction, due to an access to an IO port, or due to a memory access.

3.4 Enhancing Dependability

Commodity operating systems continue to employ system construction techniques that favor performance over dependability [29]. If their authors intend to improve system dependability, they face the challenge of enhancing the large existing device driver base, potentially without source code access to all drivers.

Our architecture improves system availability and reliability, while avoiding modifications to the device drivers, via driver isolation within virtual machines. The VM provides a hardware protection domain, deprives the driver, and inhibits its access to the remainder of the system (while also protecting the driver from the rest of the system). The use of the virtual machine supports today's systems and is practical in that it avoids a large engineering effort.

The device driver isolation helps to improve *reliability* by preventing fault propagation between independent components. It improves driver *availability* by supporting fine grained driver restart (virtual machine reboot). Improved driver availability leads to increased system reliability when clients of the drivers promote fault containment. Proactive restart of drivers, to reset latent errors or to upgrade drivers, reduces dependence on recursive fault containment, thus helping to improve overall system reliability.

The DD/OS solution supports a continuum of configurations for device driver isolation, from individual driver isolation within dedicated VMs to grouping of all drivers within a single VM. Grouping drivers within the same DD/OS reduces the availability of the DD/OS to that of the least stable driver (if not further). Even with driver grouping, the system enjoys the benefits of fault isolation and driver restart.

Driver restart is a response to one of two event types: asynchronous (e.g., in response to fault detection [33], or in response to a malicious driver), or synchronous (e.g., live upgrades [23] or proactive restart [5]). The reboot response to driver failure returns the driver to a known good state: its initial state. The synchronous variant has the advantage of being able to quiesce the DD/OS prior to rebooting, and to negotiate with clients to complete sensitive tasks. Our solution permits restart of any driver via a VM reboot. However, drivers that rely on a hardware reset to reinitialize their devices may not be able to recover their devices.

The interface between the DD/OS and its clients provides a natural layer of indirection to handle the discontinuity in service due to restarts. The indirection *captures* accesses to a restarting driver. The access is either delayed until the connection is transparently restarted [23] (requiring the DD/OS or the VMM to preserve canonical cached client state across the restart), or reflected back to the client as a fault.

4 Virtualization Issues

The isolation of the DD/OS via a virtual machine introduces several issues: the DD/OS consumes resources beyond those that a device driver requires, it performs DMA operations, and it can violate the special timing needs of physical hardware. Likewise, legacy operating systems are not designed to collaborate with other operating systems to control the devices within the system. This section presents solutions to these issues.

4.1 DMA Address Translation

DMA operates on physical addresses of the machine. In a VM, memory addresses are subject to another address translation: from guest physical to host physical addresses. Since devices are not subject to TLB address translation, DMA addresses calculated inside the VM and fed to a hardware device reference incorrect host memory addresses.

Virtual machine monitors usually run device drivers at kernel privilege level [3, 21, 35]. The VMM exports virtual hardware devices to the VM, which may or may not resemble the real hardware in the system. On device access the monitor intercepts and translates requests and

DMA addresses to the machine's real hardware. Since all hardware accesses including DMA requests are intercepted, the VM is confined to its compartment.

When giving a VM unrestricted access to DMA-capable devices, the VM-to-host memory translation has to either be incorporated into all device requests or the DMA address translation has to be preserved. The particular approach depends on available hardware features and the virtualization method (full virtualization vs. paravirtualization).

In a paravirtualized environment the DD/OS can incorporate the VMM page mappings into the DMA address translation. For the Linux kernel this requires modification to only a few functions. The hypervisor also has to support an interface for querying and pinning the VM's memory translations.

When DMA address translation functions can't be overridden, the DD/OS's have to be mapped idempotently to physical memory. Apparently, this would restrict the system to a single DD/OS instance. But by borrowing ideas from single-address-space OS's we can overcome this restriction under certain circumstances. In many cases device drivers only issue DMA operations on dynamically allocated memory, such as the heap or page pool. Hence, only those pages require the restriction of being mapped idempotently. Using a memory balloon driver [36], pages can be reclaimed for use in other DD/OS's, effectively sharing DMA-capable pages between all DD/OS's (see Figure 2).

DMA from static data pages, such as microcode for SCSI controllers, further requires idempotent mapping of data pages. However, dynamic driver instantiation usually places drivers into memory allocated from the page pool anyway. Alternatively, one DD/OS can run completely unrellocated; multiple instances of the same OS can potentially share the read-only parts.

It is important to note that all solutions assume well-behaving DD/OS's. Without special hardware support, DD/OS's can still bypass memory protection by performing DMA to physical memory outside their compartments.

4.2 DMA and Trust

Code with unrestricted access to DMA-capable hardware devices can circumvent standard memory protection mechanisms. A malicious driver can potentially elevate its privileges by using DMA to replace hypervisor code or data. In any system without explicit hardware support to restrict DMA accesses, we have to consider device drivers as part of the trusted computing base.

Isolating device drivers in separate virtual machines can still be beneficial. Nooks [33] only offers very weak protection by leaving device drivers fully privileged, but

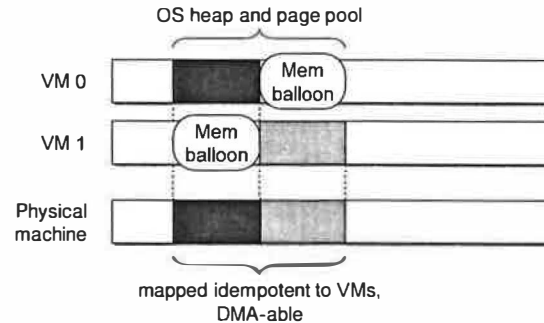


Figure 2: DMA memory allocation for two VMs. The balloon driver enables reallocation of the memory.

still reports a successful recovery rate of 99% for synthetically injected driver bugs. The fundamental assumption is that device drivers may fault, but are not malicious.

We differentiate between three trust scenarios. In the first scenario only the client of the DD/OS is untrusted. In the second case both the client as well as the DD/OS are untrusted by the hypervisor. In the third scenario the client and DD/OS also distrust each other. Note that the latter two cases can only be enforced with DMA restrictions as described in the next section.

During a DMA operation, page translations targeted by DMA have to stay constant. If the DD/OS's memory is not statically allocated it has to explicitly pin the memory. When the DD/OS initiates DMA in or out of the client's memory to eliminate copying overhead, it must pin that memory as well. In the case that the DD/OS is untrusted, the hypervisor has to enable DMA permissions to the memory and to ensure that the DD/OS cannot run denial-of-service attacks by pinning excessive amounts of physical memory.

When the DD/OS and client distrust each other, further provisions are required. If the DD/OS gets charged for pinning memory, a malicious client could run a DoS attack against the driver. A similar attack by the DD/OS against the client is possible when the DD/OS performs the pinning on behalf of the client. The solution is a co-operative approach with both untrusted parties involved. The client performs the pin operation on its own memory, which eliminates a potential DoS attack by the DD/OS. Then, the DD/OS validates with the hypervisor that the pages are sufficiently pinned. By using time-bound pinning [27] guaranteed by the hypervisor, the DD/OS can safely perform the DMA operation.

Page translations also have to stay pinned during a VM restart, since a faulting DD/OS may leave a device actively using DMA. All potentially targeted memory thus cannot be reclaimed until the VMM is sure that outstanding DMA operations have either completed or aborted.

Likewise, client OS's must not use memory handed out to the faulted DD/OS until its restart has completed.

4.3 IO-MMU and IO Contexts

The IO-MMU, initially designed to overcome the 32-bit address limitation for DMA in 64-bit systems, enables remapping bus addresses to host addresses at page granularity. IO-MMUs are, amongst others, available in AMD Opteron [1], Alpha 21172 [8], and HP Itanium systems [22]. They can be used to enforce access permissions for DMA operations and to translate DMA addresses. Thus, DD/OS's can be fully hardware-isolated from the VMM and other VMs, removing device drivers from the trusted computing base [24].

Tailored towards monolithic OS designs, IO-MMUs usually don't support multiple address contexts, such as per device, per slot, or per bus translations. The conflicting sets of virtual to physical mappings of isolated device drivers prevent simultaneous use of these IO-MMUs. We emulate multiple IO address contexts by time-multiplexing the IO-MMU between PCI devices. Resembling task scheduling, we periodically schedule IO-MMU contexts and enable bus access for only those devices that are associated with the active context.

The PCI specification [30] does not define a maximum access latency to the PCI bus, but only requires fair arbitration preventing deadlocks. Devices therefore have to be designed for potentially long bus access latencies—up to multiple milliseconds—which makes a coarse-grained scheduling approach feasible. The scheduling period has to be within the bounds of each device's timing tolerances; the particular handling of timeouts is specific to the device class. For example network cards simply start dropping packets when the card's internal buffers overflow, whereas the IDE DMA controller signals an error condition.²

A downside of time multiplexing is that the average available bus bandwidth for a device decreases and delivery latency increases. Benchmarks with a gigabit Ethernet NIC show a throughput decrease that is proportional to the allocated bus share. We further reduce the impact

²IO-MMU time multiplexing is not fully transparent for all device classes. For example, the IDE DMA controller in our experimental AMD Opteron system requires dedicated handling. The IDE controller's behavior changes based on its DMA state: DMA startup or in-progress DMA. For DMA startup it can accept a multi-millisecond latency until its first bus access is permitted to proceed. But if its bus master access is rescinded for a multi-millisecond duration during an active DMA operation, it aborts instead of retrying the operation. The problem is that the millisecond scheduling period exceeds the device's latency. We therefore additionally check for in-progress DMA directly at the IDE controller and delay the preemption until DMA completion. However, to perform this test we need specific device knowledge—even though it is for a whole device class—compromising the transparency of our approach.

of time multiplexing by dynamically adapting bus allocations based on device utilization, preferring active and asynchronously operating devices.

The IO-MMU time multiplexing is a performance compromise to support device driver isolation on inadequate hardware, and is a proof-of-concept for our reuse and isolation goals. Future hardware solutions could eliminate the need for time multiplexing.

4.4 Resource Consumption

Each DD/OS consumes resources that extend beyond the inherent needs of the driver itself. The DD/OS needs a minimum amount of memory for code and data. Furthermore, each DD/OS has a certain dynamic processing overhead for periodic timers and housekeeping, such as page aging and cleaning. Periodic tasks in DD/OS's lead to cache and TLB footprints, imposing overhead on the clients even when not using any device drivers.

Page sharing as described in [36] significantly reduces the memory and cache footprint induced by individual DD/OS's. The sharing level can be very high when the same DD/OS kernel image is used multiple times and customized with loadable device drivers. In particular, the steady-state cache footprint of concurrent DD/OS's is reduced since the same housekeeping code is executed. It is important to note that memory sharing not only reduces overall memory consumption but also the cache footprint for physically tagged caches.

The VMM can further reduce the memory consumption of a VM by swapping unused pages to disk. However, this approach is infeasible for the DD/OS running the swap device itself (and its dependency chain). Hence, standard page swapping is permitted to all but the swap DD/OS. When treating the DD/OS as a black box, we cannot swap unused parts of the swap DD/OS via working set analysis. All parts of the OS must always be in main memory to guarantee full functionality even for rare corner cases.

Besides memory sharing and swapping, we use three methods to further reduce the memory footprint. Firstly, memory ballooning actively allocates memory in the DD/OS, leading to self-paging [18, 36]. The freed memory is handed back to the VMM. Secondly, we treat zero pages specially since they can be trivially restored. Finally, we compress [6] the remaining pages that do not belong to the active working set and that are not safe to swap, and uncompress them on access.

Page swapping and compression are limited to machines with DMA hardware that can fault on accesses to unmapped pages. Otherwise, a DMA operation could access invalid data (it must be assumed that all pages of a DD/OS are pinned and available for DMA when treating the DD/OS as a black box).

Periodic tasks like timers can create a non-negligible steady-state runtime overhead. In some cases the requirements on the runtime environment for a DD/OS whose sole purpose is to encapsulate a device driver can be weakened in favor of less resource consumption. For example, a certain clock drift is acceptable for an idle VM as long as it does not lead to malfunction of the driver itself, allowing us to schedule OS's less frequently or to simply drop their timer ticks.

4.5 Timing

Time multiplexing of multiple VMs can violate timing assumptions made in the operating system code. OS's assume linear time and non-interrupted execution. Introducing a virtual time base and slowing down the VM only works if there is no dependence on real time. Hardware devices, however, are not subject to this virtual time base. Violating the timing assumptions of device drivers, such as short delays using busy waiting or bound response times, can potentially lead to malfunctioning of the device.³

We use a scheduling heuristic to avoid preemption within time critical sections, very similar to our approach to lock-holder preemption avoidance described in [34]. When consecutive operations are time-bound, operating systems usually disable preemption, for example by disabling hardware interrupts. When the VMM scheduler would preempt a virtual processor but interrupts are disabled, we postpone the preemption until interrupts are re-enabled, thereby preserving the timing assumptions of the OS. This requires the VMM to trap the re-enable operation. Hard preemption after a maximum period avoids potential DoS attacks by malicious VMs.

4.6 Shared Hardware and Recursion

Device drivers assume exclusive access to the hardware device. In many cases exclusiveness can be guaranteed by partitioning the system and only giving device access to a single DD/OS. Inherently shared resources, such as the PCI bus and PCI configuration space, are incompatible with partitioning and require shared and synchronized access for multiple DD/OS's. Following our reuse approach, we give one DD/OS full access to the shared device; all other DD/OS's use driver stubs to access the shared device. The server part in the controlling DD/OS can then apply a fine-grained partitioning policy. For example, our PCI DD/OS partitions devices based on a

³Busy waiting, which relies on correct calibration at boot time, is particularly problematic when the calibration period exceeds a VM scheduling time slice and thus reports a slower processor. A device driver using busy waiting will then undershoot a device's minimal timing requirements.

configuration file, but makes PCI bridges read-only accessible to all client DD/OS's. To simplify VM device discovery, additional virtual devices can be registered.

In a fully virtualized environment, some device drivers cannot be replaced dynamically. Linux, for example, does not allow substituting the PCI bus driver. In those cases, full hardware emulation is required by the VMM. The number of such devices is quite limited. In the case of Linux the limitations include PCI, the interrupt controller, keyboard, mouse, and real-time clock.

5 Evaluation

We implemented a driver reuse system according to the architecture described in the prior sections, and assessed the architecture's performance, resource, and engineering costs. We evaluated reused drivers for the network, disk and PCI subsystems. We limit our evaluation to a paravirtualization environment.

To support a comparative performance analysis, we constructed a baseline system and a device driver reuse system that closely resemble each other. They use identical device driver code. They run the same benchmarks, utilizing the same protocol stacks and the same OS infrastructure. They differ in their architectures: the baseline uses its native device driver environment, while our system uses the driver reuse environment and is paravirtualized. The baseline OS is a standard Linux operating system. The device driver reuse system is constructed from a set of paravirtualized Linux OS's configured as DD/OS components and client components. The client OS communicates with the reused device drivers via special kernel modules installed into the client OS.

5.1 Virtualization Environment

The paravirtualization environment is based on the L4 microkernel [25]. L4 serves as a small privileged-mode hypervisor. It offers minimal abstractions and mechanisms to support isolation and communication for the virtual machines. Fewer than 13,000 lines of code run privileged.

The DD/OS and the client OS are provided by two different generations of the Linux kernel: versions 2.4.22 and 2.6.8.1. The 2.4 kernel was ported to the L4 environment in the tradition of the original L4Linux adaptation [19]. In contrast, we used a very lightweight adaptation of the 2.6 kernel to L4, with roughly 3000 additional lines of code (and only 450 lines intrusive). The paravirtualized Linux kernels use L4 mechanisms to receive interrupts, to schedule, to manage application memory, and to handle application system calls and exceptions.

The VMM, a user-level L4 task, coordinates resources

such as memory, device mappings, and I/O port mappings for the DD/OS instances and the client OS.

All components communicate via L4 mechanisms. These mechanisms include the ability to establish shared pages, perform high-speed IPC, and to efficiently copy memory between address spaces. The mechanisms are coordinated by object interfaces defined in a high-level IDL, which are converted to optimized inlined assembler with an IDL compiler.

5.2 Translation Modules

For efficient data transfer, the client and DD/OS communicate enough information to support DMA directly from the client's pages via a shared producer-consumer command ring. In a typical sequence, the client adds device commands to the ring and activates the DD/OS via a virtual interrupt, and then the DD/OS services the command. Before performing the device DMA operation, the DD/OS validates the legality of the client's addresses and the client's pinning privileges.

The DD/OS does not generate virtual addresses for the client's pages; Linux device drivers are designed to support DMA operations on pages that are not addressable within the Linux kernel's virtual address space (by default, Linux can only address about 940MB of memory in its kernel space). The Linux drivers refer to pages indirectly via a page map. To leverage Linux's page map, we configure Linux with knowledge of all physical pages on the machine, but reserved from use (any attempts to access memory outside the DD/OS's VM causes page permission faults), and then convert client request addresses into page map offsets. In case a driver or subsystem places restrictions on acceptable addresses, it may be necessary to first copy the data.

Disk Interface The disk interface communicates with Linux's block layer, and is added to the DD/OS as a kernel module. It converts client disk operations into Linux block requests, and injects the block requests into the Linux kernel. Linux invokes the translation layer upon completion of the requests via a callback associated with each request. The block layer additionally supports the ability for the DD/OS to process requests out-of-order. The client and DD/OS share a set of request ID's to identify the reordered commands.

Network Interface The network interface has the additional feature of asynchronous inbound packet delivery. We developed our system to support multiple clients, and thus the DD/OS accepts the inbound packets into its own memory for demultiplexing. While outbound packets are transmitted from the client via DMA, inbound packets are securely copied from the DD/OS to the client

by the L4 microkernel, thus protecting the client memory from the DD/OS (and requires agreement from the client to receive the packets). The L4 kernel creates temporary CPU-local memory mappings, within the L4 kernel space, to achieve an optimized copy.

The translation layer is added to the DD/OS as a device driver module. It represents itself to the DD/OS as a Linux network device, attached to a virtual interconnect. But it doesn't behave as a standard network device; instead it appends outbound packets directly to the real adapter's kernel packet queue (in the manner of network filters), where they are automatically rate controlled via the real device's driver feedback to the Linux kernel.

To participate directly on the physical network, the translation layer accepts inbound packets using the Linux ISO layer-two bridging module hook. The translation layer queues the packets to the appropriate client OS, and eventually copies to the client.⁴

PCI Interface When the PCI driver is isolated, it helps the other DD/OS instances discover their appropriate devices on the bus, and restricts device access to only the appropriate DD/OS instances.

The PCI interface is not performance critical. We forward all client PCI configuration-space read and write requests to the PCI DD/OS. It will perform write requests only for authorized clients. For read requests, it provides accurate information to the device's DD/OS, and contrived information to other clients.

We execute the PCI DD/OS at a lower priority than all other system components. With no timing requirements, it can tolerate severe clock drift.

5.3 Resource Consumption

For memory, we measured the active and steady-state page working set sizes of DD/OS instances, and considered the effect of page sharing and memory compression for all pages allocated to the DD/OS instances. For CPU, we focused on the idle cycle consumption (later sections explore the CPU costs of active workloads).

To avoid unnecessary resource consumption in the DD/OS, we configured the Linux kernel, via its build configuration, to include only the device drivers and functionality essential to handle the devices intended to be used in the benchmarks. The runtime environment of each DD/OS is a tiny ROM image which initializes into a single-user mode with almost no application presence.

⁴An alternative to packet copying, page remapping, has a prohibitively expensive TLB flush penalty on SMPs when maintaining TLB coherence. A future alternative is to use a spare hyperthread to copy the packets. If the network DD/OS has only a single client, then the client can provide the pages backing the inbound packets, avoiding the copy.

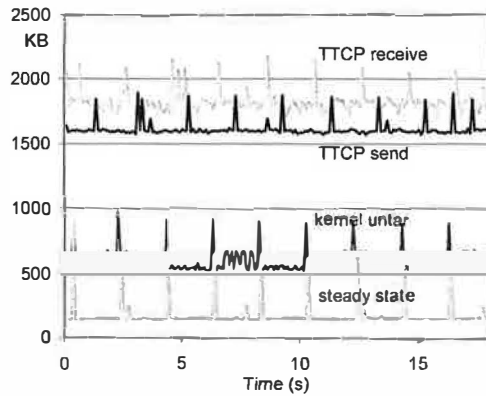


Figure 3: 90ms aggregate samples of Linux 2.6.8.1 DD/OS memory working sets when idle and for various disk and network benchmarks.

The data was collected while using Linux 2.6.8.1. The numbers are generally similar for Linux 2.4.22.

Working Set Figure 3 is a plot of memory page working sets of disk and network DD/OS's, where each sample covers 90ms of events. The “steady state” graph shows the inherent fixed cost of an idle DD/OS, usually around 144KB, with a housekeeping spike about every two seconds. The remaining graphs provide an idea of working set sizes during activity. The “tcp receive” and “tcp send” tests show the working set sizes during intense network activity. The “untar” test shows the working set response to the process of unarchiving a Linux kernel source tree to disk. The worst-case working set size reaches 2200KB, corresponding to high network activity. Our configuration is susceptible to a large working set for network activity because the DD/OS buffers incoming packets within its own memory. However, due to Linux's reuse of packet buffers the DD/OS working set size remains bounded.

Memory Compression To test the possibility of sharing and compressing the pages that back the DD/OS instances, we performed an offline analysis of a snapshot of a particular DD/OS configuration. The tested configuration included three DD/OS instances, one each for PCI, IDE, and the Intel e1000 gigabit. The PCI VM was configured with 12MB and the others with 20MB memory each. We ran the PostMark benchmark stressing a VM with Linux 2.6 serving files via NFS from the local IDE disk over the network. The active memory working set for all DD/OS's was 2.5MB.

For systems without an IO-MMU, the memory consumption can only be reduced by cooperative memory ballooning [36]. With the balloon driver in the DD/OS's we can reclaim 33% of the memory.

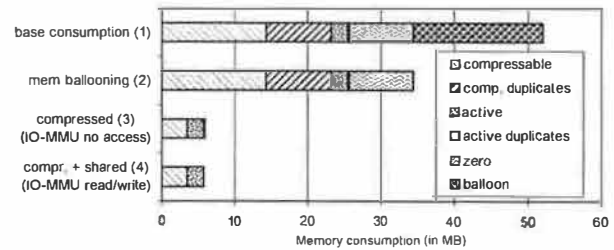


Figure 4: (1) Combined memory consumption of disk, network, and PCI DD/OS's with 20MB, 20MB, and 12MB VMs, (2) after memory ballooning, (3) with memory compression, and (4) memory compression and sharing.

Using an IO-MMU that can recover from page faults, we can revoke page access rights and compress memory that is not part of the active working set. Support of read-only page access rights by the IO-MMUs furthermore enables sharing of identical pages of the active working set via copy-on-write. We searched for duplicate pages among the three DD/OS instances. Any duplicate page is shareable whether it is in an active working set or not. A page in any DD/OS instance is additionally upgraded to an active page if it has a duplicate in any working set, to avoid having a compressed as well as an uncompressed copy. Finally, the IO-MMU enables us to reclaim all zero pages uncooperatively. For the given setup, up to 89% of the allocated memory can be reclaimed, reducing the overall memory footprint of three concurrent DD/OS's to 6MB (see Figure 4).

Without an IO-MMU, gray-box knowledge enables DD/OS paging. For example, the memory of Linux's page map is never used for a DMA operation, and is thus pageable. Furthermore, the network and block DD/OS each had a contiguous 6.9 MB identical region in their page maps, suitable for sharing.

CPU Utilization The steady state of a DD/OS has an inherent CPU utilization cost, not just influenced by internal activities, but also by the number of DD/OS's in the system. We measured the DD/OS CPU utilization response to additional DD/OS instances; the first eight DD/OS's each consume 0.12% of the CPU, and then the ninth consumes 0.15%, and the tenth consumes 0.23% (see Figure 5).

The DD/OS's were idle with no device activity. Only the first DD/OS was attached to a device—the PCI bus. The others contained a single device driver (the e1000).

The machine was a Pentium 4 2.8 GHz with a 1MB L2 cache, which can almost fit the steady-state memory working sets of seven DD/OS instances (at 144KB each, see Figure 3). The L2 cache miss rate began to rapidly rise with the eighth DD/OS, leading to an inflection in the CPU utilization curve.

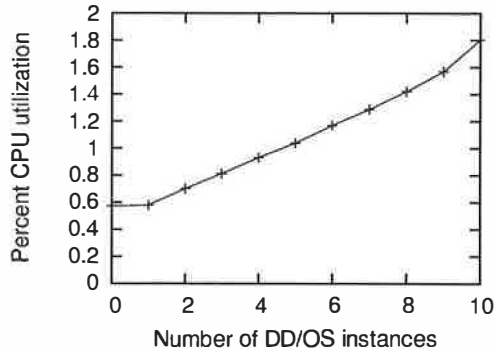


Figure 5: Incremental CPU utilization for additional steady state DD/OS instances, representing the fixed cost of executing a DD/OS.

5.4 Performance

A set of benchmarks allowed us to explore the performance costs of the DD/OS approach to device driver reuse, stressing one driver at a time, and then using network and disk drivers together. The networking benchmarks were selected to help provide a point of comparison with recent literature.

We executed our benchmarks with two device driver reuse scenarios: (1) with all drivers consolidated in single DD/OS, and (2) with the devices isolated in dedicated DD/OS instances. For a baseline, the benchmarks are also executed within the original, native device driver environment.

The benchmark OS ran Debian Sarge with the Linux 2.6 kernels, constrained to 768MB. When using the Linux 2.4 kernels, performance numbers were very similar. The hardware used in the test system was a Pentium 4 2.8 GHz processor, with an Intel 82540 gigabit network PCI card, and a desktop SATA disk (Maxtor 6Y120M0).

TTCP Figure 6 presents the throughput of the TTCP benchmark relative to the native throughput, using two packet sizes. Throughput at the 1500-byte packet size remains within 3% of native, and drops to 8% of native for 500-byte packets. Linux performs the packet sizing within the kernel, rather than within TTCP, via use of Linux's maximum transmission unit (MTU) parameter, avoiding a per-packet address space transition. The CPU utilization relative to native Linux was 1.6x for send, 2.06x for receive with 1500-byte MTU, and 2.22x for receive with 500-byte MTU. As expected, network receive generated a larger CPU load than network send due to extra packet copies. TTCP was configured for a 128KB socket buffer size.

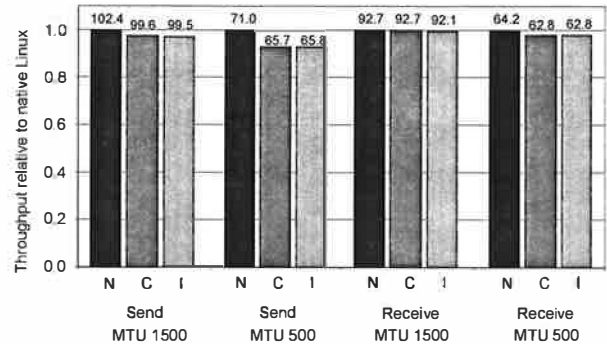


Figure 6: Normalized TTCP throughput results for native Linux (N), consolidated (C), and isolated (I) DD/OS's. Absolute throughput given in MB/s.

Netperf The Netperf benchmark confirmed the TTCP MTU 1500 results; throughput with driver reuse remained within 3% of native, with 1.6x CPU utilization for sending, and up to 2.03x CPU utilization for receiving. The native throughput was 98.5 MB/s. A substantial increase in TLB and L2 cache misses led to higher CPU utilization. These misses are inherent to our test-platform; the Pentium 4 flushes TLBs and L1 caches on every context switch between the client and DD/OS. The Netperf benchmark transferred one gigabyte, with a 32KB send and receive size, and a 256KB socket buffer size.

Disk Figure 7 presents the results of our *streaming* disk benchmark for the isolated DD/OS's (consolidated results are identical). The benchmark highlights the overhead of our solution, as opposed to masking it with random-access disk latency. The benchmark bypasses the client's buffer cache (using a Linux raw device) and file system (by directly accessing the disk partition). We thus avoid timing the behavior of the file system. Native throughput averaged 50.75 MB/s with a standard deviation of 0.46 MB/s. For driver reuse, the throughput was nearly identical and the difference less than half the standard deviation, with CPU utilization ranging from 1.2x to 1.9x native.

Application-Level We studied application-level performance with the PostMark benchmark, run over NFS. This benchmark emulates the file transaction behavior of an Internet electronic mail server, and in our scenario, the file storage is provided by an NFS server machine. The benchmark itself executes on a client machine. The NFS server used our driver reuse framework, and was configured as in the microbenchmarks. The client had a 1.4 GHz Pentium 4, 256MB memory, a 64MB Debian RAM disk, an Intel 82540 gigabit Ethernet PCI card, and executed a native Linux 2.6.8.1 kernel. The performance of

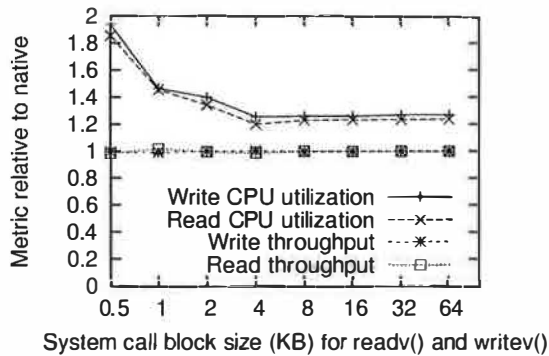


Figure 7: Throughput and CPU use relative to native Linux for disk streaming read and write.

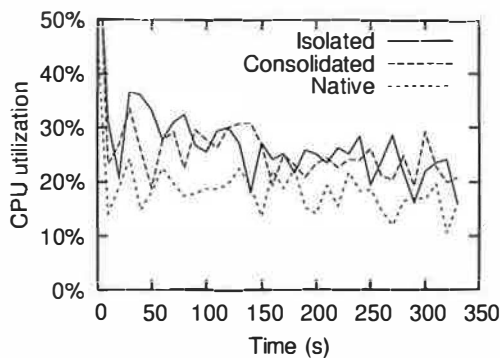


Figure 8: CPU utilization for the NFS server machine while handling the PostMark benchmark.

the NFS server was nearly identical for all driver scenarios, for native Linux and for driver reuse, with an average runtime of 343.4 seconds. The standard deviation, 2.4%, was over twice the loss in performance for driver reuse. Both the isolated and consolidated driver reuse configurations had higher CPU utilization than native Linux; see Figure 8 for CPU utilization traces of the NFS server machine covering the duration of the benchmark. The benchmark starts with a large CPU spike due to file creation. Postmark was configured for file sizes ranging from 500-bytes to 1MB, a working set of 1000 files, and 10000 file transactions.

5.5 IO-MMU

We used a 1.6 GHz AMD Opteron system with an AMD 8111 chipset to evaluate IO-MMU time multiplexing. The chipset's graphics aperture relocation table mechanism relocates up to 2GB of the 4GB DMA space at a 4KB granularity [1]. The chipset only supports read-write and no-access permissions.

Each virtual machine running a DD/OS has a dedicated IO-MMU page table which is synchronized with

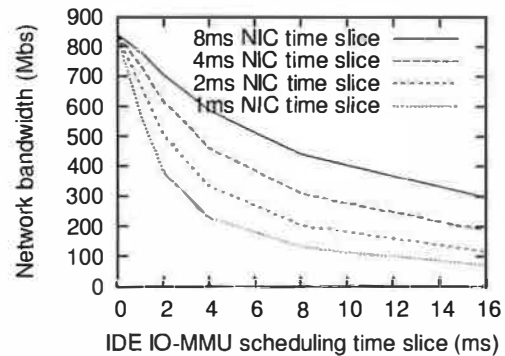


Figure 9: Network bandwidth in response to various IO-MMU context scheduling rates.

the guest-physical-to-host-physical mappings of the VM. When clients grant the DD/OS access to parts of their memory, appropriate entries are added to the IO-MMU page table as well.

The VMM connects the managed PCI devices of each DD/OS with their respective IO-MMU contexts. Periodically, but independent of the processor scheduler, we switch between the IO contexts. On context switch, the hypervisor enables and disables bus master access in the PCI configuration space for the respective devices. Our shortest scheduling granularity of 1ms is limited by the frequency of the periodic timer.

We evaluated the performance and overhead of scheduling IO-MMU contexts, as well as the bounds of the scheduling period for hardware devices. The test system contained two DD/OS's, one driving an Intel e1000 gigabit Ethernet adapter and the other handling the IDE disk controller.

First, and most importantly, we can completely isolate the physical memory covered by the IO-MMU and transparently relocate both VMs. Neither VM is able to perform DMA to memory outside its compartment. We ran the TTCP benchmark and varied the bus allocation for the NIC and disk controller. The network throughput scaled almost linearly with the bus allocation. The NIC started dropping packets when it lost access to the bus for more than 8ms. Figure 9 shows the achieved network bandwidth for various scheduling configurations.

The IDE controller is less bandwidth-sensitive since the throughput is bounded by disk latency. However, our scheduling granularity of 1ms exceeds the timeout for in-progress transactions. When disabling bus master we therefore postpone IDE deactivation when operations are still in-flight. The overhead for IO-MMU context switching was a 1% increase in CPU utilization.

	server	client	common	total
network	1152	770	244	2166
block 2.4	805	659	108	1572
block 2.6	751	546	0	1297
PCI	596	209	52	857
common	0	0	620	620
total	3304	2184	1024	

Figure 10: Itemization of source lines of code used to implement our evaluation environment. Common lines are counted once.

5.6 Engineering Effort

We estimate engineering effort in man hours and in lines of code. The translation modules and client device drivers for the block and network, along with the user-level VMM, were written by a single student over roughly a two month period, originally for L4Linux 2.4. This student already had experience with Linux network driver development for a paravirtualized Linux on L4. A second student implemented the PCI support within one week.

The 2.4 network translation module was easily upgraded to serve as the translation module for Linux 2.6, with minor changes. However the 2.4 block translation module was mostly incompatible with 2.6's internal API (Linux 2.6 introduced a new block subsystem). We thus wrote new block translation and client device drivers for 2.6. We successfully reused the 2.6 block and network drivers with the 2.4 client, and vice versa.

See Figure 10 for an itemization of the lines of code. The figure distinguishes between lines specific to the translation modules added to the server, lines specific to the virtual device drivers added to the client, and additional lines that are common (and are counted once).

The achieved code reuse ratio is 99.9% for NIC drivers in Linux; the translation modules add 0.1% to their code base. When we additionally include all code required for the virtualization—the L4 microkernel, the VMM, and the paravirtualization modifications—we still achieve a reuse ratio of 91% just for Linux's NIC driver base.

The engineering effort enabled us to successfully reuse Linux device drivers with all of our tested lab hardware. The following drivers were tested: Intel gigabit, Intel 100 Mbit, Tulip (with a variety of Tulip compatible hardware), Broadcom gigabit, pcnet32, ATA and SATA IDE, and a variety of uniprocessor and SMP chipsets for Intel Pentium 3/4 and AMD Opteron processors.

6 Discussion and Future Work

We presented a new approach to reusing unmodified device drivers and enhancing system dependability using virtual machines, but evaluated only a paravirtual-

ized implementation. Paravirtualization is an enhanced machine API that relocates some functionality from the guest OS to the VMM and hypervisor [16]. For example, it permits our DD/OS instances to directly translate their virtual addresses into bus addresses for DMA. It also provides performance benefits [3, 16] compared to use of the real machine API. We have discussed the issues related to device driver pass-through with full virtualization, and consider our paravirtualization implementation to be an approximation. In terms of correctness, the primary difference relates to proper address translation for DMA operations, which becomes irrelevant with hardware device isolation (such as the IO-MMU). In terms of performance, the paravirtualization numbers underestimate the costs of a fully-virtualized solution.

Our system currently supports a sufficiently large subset of device classes to be self-hosting in a server environment. We have not addressed the desktop environment, which requires support for the graphics console, USB, Firewire, etc.

Generic driver stubs only provide access to the least common denominator, thereby hiding more advanced hardware features. Our client-server model enables device access at any level in the software hierarchy of the DD/OS, even allowing programming against richer OS APIs like TWAIN, or enabling vendor-specific features such as DVD burning. Using the appropriate software engineering methods, e.g., an IDL compiler, one can quickly generate cross-address-space interfaces that support APIs with rich feature sets.

7 Conclusion

Widely used operating systems support a variety of devices; for example, in Linux 2.4 on IA32, 70% of 1.6 million lines of kernel code implement device support [7]. New operating system endeavors have the choice of either leveraging the existing device drivers, or expending effort to replicate the driver base. We present a technique that enables unmodified reuse of the existing driver base, and most importantly, does so in a manner that promotes independence of the new OS endeavor from the reused drivers.

The driver independence provides an opportunity to improve system dependability. The solution fortifies the reused drivers (to the extent supported by hardware) to promote enhanced reliability and availability (with independent driver restart).

Our method for reusing unmodified drivers and improving system dependability via virtual machines achieves good performance. For networking, where packetized throughput is latency-sensitive, the throughput remains within 3–8% of the native system. The driver isolation requires extra CPU utilization, which can be re-

duced with hardware acceleration (such as direct DMA for inbound packets).

The DD/OS solution is designed for minimal engineering effort, even supporting reuse of binary drivers. The interface implementation between the new OS and reused drivers constitutes a trivial amount of code, which leverages the vast world of legacy drivers. Driver source code, by design, remains unmodified.

References

- [1] Advanced Micro Devices, Inc. *BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors*, Apr. 2004.
- [2] J. Appavoo, M. Auslander, D. DaSilva, D. Edelsohn, O. Krieger, M. Ostrowski, et al. Utilizing Linux kernel components in K42. Technical report, IBM Watson Research, Aug. 2002.
- [3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, et al. Xen and the art of virtualization. In *Proc. of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, Oct. 2003.
- [4] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, Oct. 1997.
- [5] G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Eighth IEEE Workshop on Hot Topics in Operating Systems*, Schloss Elmau, Germany, May 2001.
- [6] R. Cervera, T. Cortes, and Y. Becerra. Improving application performance through swap compression. In *Unix Annual Technical Conference*, Monterey, CA, June 1999.
- [7] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. In *Proc. of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, Oct. 2001.
- [8] Digital Equipment Corporation. *Digital Semiconductor 21172 Core Logic Chipset, Technical Reference Manual*, Apr. 1996.
- [9] K. Elphinstone and S. Götz. Initial evaluation of a user-level device driver framework. In *9th Asia-Pacific Computer Systems Architecture Conference*, Beijing, China, Sept. 2004.
- [10] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, Oct. 1997.
- [11] A. Forin, D. Golub, and B. Bershad. An I/O system for Mach 3.0. In *Proc. of the Second USENIX Mach Symposium*, Monterey, CA, Nov. 1991.
- [12] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Reconstructing I/O. Technical Report UCAM-CL-TR-596, University of Cambridge, Computer Laboratory, Aug. 2004.
- [13] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, Boston, MA, Oct. 2004.
- [14] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, et al. The SawMill multiserver approach. In *9th SIGOPS European Workshop*, Kolding, Denmark, Sept. 2000.
- [15] S. Goel and D. Duchamp. Linux device driver emulation in Mach. In *USENIX Annual Technical Conference*, San Diego, CA, Jan. 1996.
- [16] R. P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7(6), 1974.
- [17] D. B. Golub, G. G. Sotomayor, Jr., and F. L. Rawson III. An architecture for device drivers executing as user-level tasks. In *Proc. of the USENIX Mach III Symposium*, Sante Fe, NM, Apr. 1993.
- [18] S. M. Hand. Self-paging in the Nemesis operating system. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation*, New Orleans, LA, Feb. 1999.
- [19] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of microkernel-based systems. In *Proc. of the 16th ACM Symposium on Operating System Principles*, Saint-Malo, France, Oct. 1997.
- [20] H. Härtig, J. Löser, F. Mehnert, L. Reuther, M. Pohlack, and A. Warg. An I/O architecture for microkernel-based operating systems. Technical Report TUD-FI03-08-Juli-2003, TU Dresden, Dresden, Germany, July 2003.
- [21] J. Honeycutt. *Microsoft Virtual PC 2004 Technical Overview*. Microsoft, Nov. 2003.
- [22] HP Technical Computing Division. *HP zx1 mio ERS, Rev. 1.0*. Hewlett Packard, Mar. 2003.
- [23] K. Hui, J. Appavoo, R. Wisniewski, M. Auslander, D. Edelsohn, B. Gamsa, et al. Position summary: Supporting hot-swappable components for system software. In *Eighth IEEE Workshop on Hot Topics in Operating Systems*, Schloss Elmau, Germany, May 2001.
- [24] B. Leslie and G. Heiser. Towards untrusted device drivers. Technical Report UNSW-CSE-TR-0303, School of Computer Science and Engineering, UNSW, Mar. 2003.
- [25] J. Liedtke. On μ -kernel construction. In *Proc. of the 15th ACM Symposium on Operating System Principles*, Copper Mountain Resort, CO, Dec. 1995.
- [26] J. Liedtke, U. Bartling, U. Beyer, D. Heinrichs, R. Ruland, and G. Szalay. Two years of experience with a μ -kernel based OS. *ACM SIGOPS Operating Systems Review*, 25(2), Apr. 1991.
- [27] J. Liedtke, V. Uhlig, K. Elphinstone, T. Jaeger, and Y. Park. How to schedule unlimited memory pinning of untrusted processes, or, provisional ideas about service-neutrality. In *7th Workshop on Hot Topics in Operating Systems*, Rio Rico, AR, Mar. 1999.
- [28] K. V. Maren. The Fluke device driver framework. Master's thesis, University of Utah, Dec. 1999.
- [29] D. A. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, et al. Recovery-Oriented Computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB/CS-D-02-1175, U.C. Berkeley Computer Science, Mar. 2002.
- [30] PCI Special Interest Group. *PCI Local Bus Specification, Rev. 2.1*, June 1995.
- [31] D. S. Ritchie and G. W. Neufeld. User level IPC and device management in the Raven kernel. In *USENIX Microkernels and Other Kernel Architectures Symposium*, San Diego, CA, Sept. 1993.
- [32] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *Proc. of the 2001 USENIX Annual Technical Conference*, Boston, MA, June 2001.
- [33] M. Swift, B. Bershad, and H. Levy. Improving the reliability of commodity operating systems. In *Proc. of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, Oct. 2003.
- [34] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proc. of the 3rd Virtual Machine Research and Technology Symposium*, San Jose, CA, May 2004.
- [35] VMware. *VMware ESX Server I/O Adapter Compatibility Guide*, Jan. 2003.
- [36] C. Waldspurger. Memory resource management in VMware ESX Server. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

Microreboot – A Technique for Cheap Recovery

George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, Armando Fox
Computer Systems Lab, Stanford University

{candea, skawamo, fjk, gregjf, fox}@cs.stanford.edu

Abstract

A significant fraction of software failures in large-scale Internet systems are cured by rebooting, even when the exact failure causes are unknown. However, rebooting can be expensive, causing nontrivial service disruption or downtime even when clusters and failover are employed. In this work we use separation of process recovery from data recovery to enable microrebooting – a fine-grain technique for surgically recovering faulty application components, without disturbing the rest of the application.

We evaluate microrebooting in an Internet auction system running on an application server. Microreboots recover most of the same failures as full reboots, but do so an order of magnitude faster and result in an order of magnitude savings in lost work. This cheap form of recovery engenders a new approach to high availability: microreboots can be employed at the slightest hint of failure, prior to node failover in multi-node clusters, even when mistakes in failure detection are likely; failure and recovery can be masked from end users through transparent call-level retries; and systems can be rejuvenated by parts, without ever being shut down.

1 Introduction

In spite of ever-improving development processes and tools, all production-quality software still has bugs; most of the bugs that escape testing are difficult to track down and resolve, and they take the form of Heisenbugs, race conditions, resource leaks, and environment-dependent bugs [14, 36]. Moreover, up to 80% of bugs that manifest in production systems have no fix available at the time of failure [43]. Fortunately, it is mostly application-level failures that bring down enterprise-scale software [32, 13, 35, 36], while the underlying platform (hardware and operating system) is reliable, by comparison. This is in contrast to smaller-scale systems, such as desktop computers, where hardware and operating system-level problems are still significant causes of downtime.

When failure strikes large-scale software systems, such as the ones found in Internet services, operators cannot afford to run real-time diagnosis. Instead, they focus on bringing the system back up by all means, and then do the diagnosis later. Our challenge is to find a simple, yet practical and effective approach to managing failure in large, complex systems, an approach that is accepting of the fact that bugs in application software will not be eradicated any

time soon. The results of several studies [39, 19, 34, 12] and experience in the field [5, 35, 24] suggest that many failures can be successfully recovered by rebooting, even when the failure's root cause is unknown. Not surprisingly, today's state of the art in achieving high availability for Internet clusters involves circumventing a failed node through failover, rebooting the failed node, and subsequently reintegrating the recovered node into the cluster.

Reboots provide a high-confidence way to reclaim stale or leaked resources, they do not rely on the correct functioning of the rebooted system, they are easy to implement and automate, and they return the software to its start state, which is often its best understood and best tested state. Unfortunately, in some systems, unexpected reboots can result in data loss and unpredictable recovery times. This occurs most frequently when the software lacks clean separation between data recovery and process recovery. For example, performance optimizations, such as write-back buffer caches, open a window of vulnerability during which allegedly-persistent data is stored only in volatile memory; an unexpected crash and reboot could restart the system's processes, but buffered data would be lost.

This paper presents a practical recovery technique we call *microreboot* – individual rebooting of fine-grain application components. It can achieve many of the same benefits as whole-process restarts, but an order of magnitude faster and with an order of magnitude less lost work. We describe here general conditions necessary for microreboots to be safe: well-isolated, stateless components, that keep all important application state in specialized state stores. This way, data recovery is completely separated from (reboot-based) application recovery. We also describe a prototype microrebootable system we used in evaluating microreboot-based recovery.

The low cost of microrebooting engenders a new approach to high availability, in which microrebooting is always attempted first, as front-line recovery, even when failure detection is prone to false positives or when the failure is not known to be microreboot-curable. If the microreboot does not recover the system, but some other subsequent recovery action does, the recovery time added by the initial microreboot attempt is negligible. In multi-node clusters, a microreboot may be preferable even over node failover, because it avoids overloading non-failed nodes and preserves in-memory state. Being minimally-disruptive allows microreboots to rejuvenate a system by parts without shutting down; it also allows transparent call-level retries to mask a microreboot from end users.

The rest of this paper describes, in Section 2, a design for microrebootable software and, in Section 3, a prototype implementation. Sections 4 and 5 evaluate the prototype's recovery properties using fault injection and a realistic workload. Section 6 describes a new, simpler approach to failure management that is brought about by cheap recovery. Section 7 discusses limitations of microbooting, and Section 8 presents a roadmap for generalizing our approach beyond the implemented prototype. Section 9 presents related work, and Section 10 concludes.

2 Designing Microrebootable Software

Workloads faced by Internet services often consist of many relatively short tasks, rather than long-running ones. This affords the opportunity for recovery by reboot, because any work-in-progress lost due to rebooting represents a small fraction of requests served in a day. We set out to optimize large-scale Internet services for frequent, fine-grain rebooting, which led to three design goals: fast and correct component recovery, strongly-localized recovery with minimal impact on other parts of the system, and fast and correct reintegration of recovered components.

In earlier work we introduced and motivated crash-only software [9] – programs that can be safely crashed in whole or by parts and recover quickly every time. The high-level recipe for building such systems is to structure them as a collection of small, well-isolated components, to separate important state from the application logic and place it in dedicated state stores, and to provide a framework for transparently retrying requests issued to components that are temporarily unavailable (e.g., because they are microbooting). Here we summarize the main points of our crash-only design approach.

Fine-grain components: Component-level reboot time is determined by how long it takes for the underlying platform to restart a target component and for this component to reinitialize. A microrebootable application therefore aims for components that are as small as possible, in terms of program logic and startup time. (There are many other benefits to this design, which is why it is favored for large-scale Internet software.) While partitioning a system into components is an inherently system-specific task, developers can benefit from existing component-oriented programming frameworks, as will be seen in our prototype.

State segregation: To ensure recovery correctness, we must prevent microboots from inducing corruption or inconsistency in application state that persists across microbooting. The inventors of transactional databases recognized that segregating recovery of persistent data from application logic can improve the recoverability of both the application and the data that must persist across failures. We take this idea further and require that microrebootable applications keep *all* important state in dedicated state stores located outside the application, safeguarded behind strongly-enforced high-level APIs. Examples of such state stores include transactional databases and ses-

sion state managers [26].

Aside from enabling safe microboots, the complete separation of data recovery from application recovery generally improves system robustness, because it shifts the burden of data management from the often-inexperienced application writers to the specialists who develop state stores. While the number of applications is vast and their code quality varies wildly, database systems and session state stores are few and their code is consistently more robust. In the face of demands for ever-increasing feature sets, application recovery code that is both bug-free and efficient will likely be increasingly elusive, so data/process separation could improve dependability by making process recovery simpler. The benefits of this separation can often outweigh the potential performance overhead.

Decoupling: Components must be loosely coupled, if the application is to gracefully tolerate a microreboot (μ RB). Therefore, components in a crash-only system have well-defined, well-enforced boundaries; direct references, such as pointers, do not span these boundaries. If cross-component references are needed, they should be stored outside the components, either in the application platform or, in marshalled form, inside a state store.

Retryable requests: For smooth reintegration of microbooted components, inter-component interactions in a crash-only system ideally use timeouts and, if no response is received to a call within the allotted time frame, the caller can gracefully recover. Such timeouts provide an orthogonal mechanism for turning non-Byzantine failures into fail-stop events, which are easier to accommodate and contain. When a component invokes a currently microbooting component, it receives a `RetryAfter(t)` exception; the call can then be re-issued after the estimated recovery time t , if it is idempotent. For non-idempotent calls, rollback or compensating operations can be used. If components transparently recover in-flight requests this way, intra-system component failures and microboots can be hidden from end users.

Leases: Resources in a frequently-microbooting system should be leased, to improve the reliability of cleaning up after μ RBs, which may otherwise leak resources. In addition to memory and file descriptors, we believe certain types of persistent state should carry long-term leases; after expiration, this state can be deleted or archived out of the system. CPU execution time should also be leased: if a computation hangs and does not renew its execution lease, it should be terminated with a μ RB. If requests can carry a time-to-live, then stuck requests can be automatically purged from the system once this TTL runs out.

The crash-only design approach embodies well-known principles for robust programming of distributed systems. We push these principles to finer levels of granularity within applications, giving non-distributed applications the robustness of their distributed brethren. In the next section we describe the application of some of these design principles to the implementation of a platform for microrebootable applications.

3 A Microrebootable Prototype

The enterprise edition of Java (J2EE) [40] is a framework for building large-scale Internet services. Motivated by its frequent use for critical Internet-connected applications (e.g., 40% of the current enterprise application market [3]), we chose to add microreboot capabilities to an open-source J2EE application server (JBoss [21]) and converted a J2EE application (RUBiS [37]) to the crash-only model. The changes we made to the JBoss platform universally benefit all J2EE applications running on it. In this section we describe the details of J2EE and our prototype.

3.1 The J2EE Component Framework

A common design pattern for Internet applications is the three-tiered architecture: the presentation tier consists of stateless Web servers, the application tier runs the application per se, and the persistence tier stores long-term data in one or more databases. J2EE is a framework designed to simplify developing applications for this model.

J2EE applications consist of portable components, called Enterprise Java Beans (EJBs), and platform-specific XML deployment descriptor files. A J2EE application server, akin to an operating system for Internet services, uses the deployment information to instantiate an application's EJBs inside management containers; there is one container per EJB object, and it manages all instances of that object. The server-managed containers provide the application components with a rich set of services: thread pooling and lifecycle management, client session management, database connection pooling, transaction management, security and access control, etc. In theory, a J2EE application should be able to run on any J2EE application server, with modifications only needed in the deployment descriptors.

End users interact with a J2EE application through a Web interface, the application's presentation tier, encapsulated in a WAR – Web ARchive. The WAR component consists of servlets and Java Server Pages (JSPs) hosted in a Web server; they invoke methods on the EJBs and then format the returned results for presentation to the end user. Invoked EJBs can call on other EJBs, interact with the back-end databases, invoke other Web services, etc.

An EJB is similar to an event handler, in that it does not constitute a separate locus of control – a single Java thread shepherds a user request through multiple EJBs, from the point it enters the application tier until it returns to the Web tier. EJBs provide a level of componentization that is suitable for building crash-only applications.

3.2 Microreboot Machinery

We added a microreboot method to JBoss that can be invoked programatically from within the server, or remotely, over HTTP. Since we modified the JBoss server, microreboots can now be performed on any J2EE application; however, this is safe only if the application is crash-only. The microreboot method can be applied to one or more

EJB or WAR components. It destroys all extant instances of the corresponding objects, kills all shepherding threads associated with those instances, releases all associated resources, discards server metadata maintained on behalf of the component(s), and then reinstantiates and initializes the component(s).

The only resource we do not discard on a μ RB is the component's classloader. JBoss uses a separate class loader for each EJB to provide appropriate sandboxing between components; when a caller invokes an EJB method, the caller's thread switches to the EJB's classloader. A Java class' identity is determined both by its name and the classloader responsible for loading it; discarding an EJB's classloader upon μ RB would unnecessarily complicate the update of internal references to the microrebooted component. Preserving the classloader does not violate any of the sandboxing properties. Keeping the classloader active does not reinitialize EJB static variables upon μ RB, but this is acceptable, since J2EE strongly discourages the use of mutable static variables anyway, as this would prevent transparent replication of EJBs in clusters.

Some EJBs cannot be microrebooted individually, because EJBs might maintain references to other EJBs and because certain metadata relationships can span containers. Thus, whenever an EJB is microrebooted, we microreboot the transitive closure of its inter-EJB dependents as a group. To determine these recovery groups, we examine the EJB deployment descriptors; the information on references is typically used by J2EE application servers to determine the order in which EJBs should be deployed.

3.3 A Crash-Only Application

Although many companies use JBoss to run their production applications, we found them unwilling to share their applications with us. Instead, we converted Rice University's RUBiS [37], a J2EE/Web-based auction system that mimics eBay's functionality, into eBid – a crash-only version of RUBiS with some additional functionality. eBid maintains user accounts, allows bidding on, selling, and buying of items, has item search facilities, customized information summary screens, user feedback pages, etc.

State segregation: E-commerce applications typically handle three types of important state: long-term data that must persist for years (such as customer account activity), session data that needs to persist for the duration of a user session (e.g., shopping carts or workflow state in enterprise applications), and static presentation data (GIFs, HTML, JSPs, etc.). eBid keeps these types of state in a database, dedicated session state storage, and an Ext3FS filesystem (optionally mounted read-only), respectively.

eBid uses only two types of EJBs: entity EJBs and stateless session EJBs. An entity EJB implements a persistent application object, in the traditional OOP sense, with each instance's state mapped to a row in a database table. Stateless session EJBs are used to perform higher level operations on entity EJBs: each end user operation is implemented by a stateless session EJB interacting with several

entity EJBs. For example, there is a “place bid on item X” EJB that interacts with entity EJBs User, Item, and Bid. This mixed OO/procedural design is consistent with best practices for building scalable J2EE applications [10].

Persistent state in eBid consists of user account information, item information, bid/buy/sell activity, etc. and is maintained in a MySQL database through 9 entity EJBs: IDManager, User, Item, Bid, Buy, Category, OldItem, Region, and UserFeedback. MySQL is crash-safe and recovers fast for our datasets (132K items, 1.5M bids, 10K users). Each entity bean uses container-managed persistence, a J2EE mechanism that delegates management of entity data to the EJB’s container. This way, JBoss can provide relatively transparent data persistence, relieving the programmer from the burden of managing this data directly or writing SQL code to interact with the database. If an EJB is involved in any transactions at the time of a microreboot, they are all automatically aborted by the container and rolled back by the database.

Session state in eBid takes the form of items that a user selects for buying/selling/bidding, her userID, etc. Such state must persist on the application server for long enough to synthesize a user session from independent stateless HTTP requests, but can be discarded when the user logs out or the session times out. Users are identified using HTTP cookies. Many commercial J2EE application servers store session state in middle tier memory, in which case a server crash or EJB microreboot would cause the corresponding user sessions to be lost. In our prototype, to ensure the session state survives μ RBs, we keep it outside the application in a dedicated session state repository.

We have two options for session state storage. First, we built FastS, an in-memory repository inside JBoss’s embedded Web server. The API consists of methods for reading/writing HttpSession objects atomically. FastS illustrates how session state can be segregated from the application, yet still be kept within the same Java virtual machine (JVM). Isolated behind compiler-enforced barriers, FastS provides fast access to session objects, but only survives μ RBs. Second, we modified SSM [26], a clustered session state store with a similar API to FastS. SSM maintains its state on separate machines; isolated by physical barriers, it provides slower access to session state, but survives μ RBs, JVM restarts, as well as node reboots. The session storage model is based on leases, so orphaned session state is garbage-collected automatically.

Isolation and decoupling: Compiler-enforced interfaces and type safety provide operational isolation between EJBs. EJBs cannot name each others’ internal variables, nor are they allowed to use mutable static variables. EJBs obtain references to each other (in order to make inter-EJB method calls) from a naming service (JNDI) provided by JBoss; references may be cached once obtained. The inter-EJB calls themselves are mediated by the application server via the containers and a suite of interceptors, in order to abstract away details of remote invocation and replication in the cases when EJBs are repli-

cated for performance or load balancing reasons.

Besides preservation of state across microreboots, the segregation of session state in eBid offers recovery decoupling as well, since data shared across components by means of a state store frees the components from having to be recovered together. Such segregation also helps to quickly reintegrate recovered components, because they do not need to perform data recovery following a μ RB.

4 Evaluation Framework

To evaluate our prototype, we developed a client emulator, a fault injector, and a system for automated failure detection, diagnosis, and recovery. We injected faults in eBid and measured the recovery properties of microbooting.

We wrote a **client emulator** using some of the logic in the load generator shipped with RUBiS. Human clients are modeled using a Markov chain with 25 states corresponding to the various end user operations possible in eBid, such as *Login*, *BuyNow*, or *AboutMe*; transitioning to a state causes the client to issue a corresponding HTTP request. Inbetween successive “URL clicks,” emulated clients have independent think times based on an exponential random distribution with a mean of 7 seconds and a maximum of 70 seconds, as in the TPC-W benchmark [38]. We chose transition probabilities representative of online auction users; the resulting workload, shown in Table 1, mimics the real workload seen by a major Internet auction site [16].

User operation results mostly in...	% of all requests
Read-only DB access (e.g., browse a category)	32%
Initialization/deletion of session state (e.g., login)	23%
Exclusively static HTML content (e.g., home page)	12%
Search (e.g., search for items by name)	12%
Session state updates (e.g., select item for bid)	11%
Database updates (e.g., leave seller feedback)	10%

Table 1: Client workload used in evaluating microreboot-based recovery.

To enable automatic recovery, we implemented **failure detection** in the client emulator and placed primitive diagnosis facilities in an external recovery manager. While real end users’ Web browsers certainly do not report failures to the Internet services they use, our client-side detection mimics WAN services that deploy “client-like” end-to-end monitors around the Internet to detect a service’s user-visible failures [22]. Such a setup allows our measurements to focus on the recovery aspects of our prototype, rather than the orthogonal problem of detection and diagnosis.

We implemented two fault detectors. The first one is simple and fast: if a client encounters a network-level error (e.g., cannot connect to server) or an HTTP 4xx or 5xx error, then it flags the response as faulty. If no such errors occur, the received HTML is searched for keywords indicative of failure (e.g., “exception,” “failed,” “error”). Finally, the detection of an application-specific problem

can also mark the response as faulty (such problems include being prompted to log in when already logged in, encountering negative item IDs in the reply HTML, etc.)

The second fault detector submits in parallel each request to the application instance we are injecting faults into, as well as to a separate, known-good instance on another machine. It then compares the result of the former to the “truth” provided by the latter, flagging any differences as failures. This detector is the only one able to identify complex failures, such as the surreptitious corruption of the dollar amount in a bid. Certain tweaks were required to account for timing-related nondeterminism.

We built a recovery manager (*RM*) that performs simple **failure diagnosis** and **recovers** by: microrebooting EJBs, the WAR, or all of eBid; restarting the JVM that runs JBoss (and thus eBid as well); or rebooting the operating system. *RM* listens on a UDP port for failure reports from the monitors, containing the failed URL and the type of failure observed. Using static analysis, we derived a mapping from each eBid URL prefix to a path/sequence of calls between servlets and EJBs. The recovery manager maintains for each component in the system a score, which gets incremented every time the component is in the path originating at a failed URL. *RM* decides what and when to (micro)reboot based on hand-tuned thresholds. Accurate or sophisticated failure detection was not the topic of this work; our simplistic approach to diagnosis often yields false positives, but part of our goal is to show that even the mistakes resulting from simple or “sloppy” diagnosis are tolerable because of the very low cost of μ RBs.

The recovery manager uses a simple **recursive recovery policy** [8] based on the principle of trying the cheapest recovery first. If this does not help, *RM* reboots progressively larger subsets of components. Thus, *RM* first microreboots EJBs, then eBid’s WAR, then the entire eBid application, then the JVM running the JBoss application server, and finally reboots the OS; if none of these actions cure the failure symptoms, *RM* notifies a human administrator. In order to avoid endless cycles of rebooting, *RM* also notifies a human whenever it notices recurring failure patterns. The recovery action per se is performed by remotely invoking JBoss’s microreboot method (for EJB, WAR, and eBid) or by executing commands, such as `kill -9`, over `ssh` (for JBoss and node-level reboot).

We evaluated the availability of our prototype using a new metric, **action-weighted throughput** (T_{aw}). We view a user *session* as beginning with a login operation and ending with an explicit logout or abandonment of the site. Each session consists of a sequence of user *actions*. Each user action is a sequence of *operations* (HTTP requests) that culminates with a “commit point”: an operation that must succeed for that user action to be considered successful as a whole (e.g., the last operation in the action of placing a bid results in committing that bid to the database).

An action succeeds or fails atomically: if all operations within the action succeed, they count toward action-weighted goodput (“good T_{aw} ”); if an operation fails, all

operations in the corresponding action are marked failed, counting toward action-weighted badput (“bad T_{aw} ”). Unlike simple throughput, T_{aw} accounts for the fact that both long-running and short-running operations must succeed for a user to be happy with the service. T_{aw} also captures the fact that, when an action with many operations succeeds, it generally means the user did more work than in a short action. Figure 1 gives an example of how we use T_{aw} to compare recovery by μ RB to recovery by JVM restart.

5 Evaluation Results

We used our prototype to answer four questions about microrebooting: Are μ RBs effective in recovering from failures? Are μ RBs any better than JVM restarts? Are μ RBs useful in clusters? Do μ RB-friendly architectures incur a performance overhead? Section 6 will build upon these results to show how microrebooting can change the way we manage failures in Internet services.

We used 3GHz Pentium machines with 1GB RAM for Web and middle tier nodes; databases were hosted on Athlon 2600xp+ machines with 1.5 GB of RAM and 7200rpm 120GB disks; emulated clients ran on a variety of multiprocessor machines. All machines were interconnected by a 100 Mbps Ethernet switch and ran Linux kernel 2.6.5 with Sun Java 1.4.1 and Sun J2EE 1.3.1.

5.1 Is Microrebooting Effective?

Despite J2EE’s popularity, we were unable to find any published systematic studies of faults occurring in production J2EE systems. In deciding what faults to inject in our prototype, we relied on advice from colleagues in industry, who routinely work with enterprise applications or application servers [13, 14, 24, 32, 35, 36]. They found that production J2EE systems are most frequently plagued by deadlocked threads, leak-induced resource exhaustion, bug-induced corruption of volatile metadata, and various Java exceptions that are handled incorrectly.

We therefore added hooks in JBoss for injecting artificial deadlocks, infinite loops, memory leaks, JVM memory exhaustion outside the application, transient Java exceptions to stress eBid’s exception handling code, and corruption of various data structures. In addition to these hooks, we also used FIG [6] and FAUmachine [7] to inject low-level faults underneath the JVM layer.

eBid, being a crash-only application, has relatively little volatile state that is subject to loss or corruption – much of the application state is kept in FastS / SSM. We can, however, inject faults in the data handling code, such as the code that generates application-specific primary keys for identifying rows in the DB corresponding to entity bean instances. We also corrupt class attributes of the stateless session beans. In addition to application data, we corrupt metadata maintained by the application server, but accessible to eBid code: the JNDI repository, that maps EJB names to their containers, and the transaction method map stored in each entity EJB’s container. Finally, we corrupt

data inside the session state stores (via bit flips) and in the database (by manually altering table contents).

We perform three types of data corruption: (a) set a value to *null*, which will generally elicit a `NullPointerException` upon access; (b) set an *invalid* value, i.e., a non-null value that type-checks but is invalid from the application's point of view, such as a `userID` larger than the maximum `userID`; and (c) set to a *wrong* value, which is valid from the application's point of view, but incorrect, such as swapping IDs between two users.

After injecting a fault, we used the recursive policy described earlier to recover the system. We relied on our comparison-based failure detector to determine whether a recovery action had been successful or not; when failures were still encountered, recovery was escalated to the next level in the policy. In Table 2 we show the worst-case scenario encountered for each type of injected fault. In reporting the results, we differentiate between *resuscitation*, or restoring the system to a point from which it can resume the serving of requests for all users, without necessarily having fixed the resulting database corruption, and *recovery* – bringing the system to a state where it functions with a 100% correct database. Financial institutions often aim for resuscitation, applying compensating transactions at the end of the business day to repair database inconsistencies [36]. A \approx sign in the rightmost column indicates that additional manual database repair actions were required to achieve correct recovery after resuscitation.

Based on these results, we conclude that EJB-level or WAR-level microrebooting in our J2EE prototype is effective in recovering from the majority of failure modes seen in today's production J2EE systems (first 19 rows of Table 2). Microrebooting is ineffective against other types of failures (last 7 rows), where coarser grained reboots or manual repair are required. Fortunately, these failures do not constitute a significant fraction of failures in real J2EE systems. While certain faults (e.g., JNDI corruption) could certainly be cured with non-reboot approaches, we consider the reboot-based approach simpler, quicker, and more reliable. In the cases where manual actions were required to restore service correctness, a JVM restart presented no benefits over a component μ RB.

Rebooting is a common way to recover middleware in the real world, so for the rest of this paper we compare EJB-level microrebooting to JVM process restart, which restarts JBoss and, implicitly, eBid.

5.2 Is a Microreboot Better Than a Full Reboot?

With respect to availability, Internet service operators care mostly about how many user requests their system turns away during downtime. We therefore evaluate microrebooting with respect to this end-user-aware metric, as captured by T_{aw} . We inject faults in our prototype and then allow the recovery manager (*RM*) to recover the system automatically in two ways: by restarting the JVM process running JBoss, or by microrebooting one or more EJBs, respectively. Recovery is deemed successful when

Injected Fault	Type	Reboot level	+
Deadlock		EJB	
Infinite loop		EJB	
Application memory leak		EJB	
Transient exception		EJB	
Corrupt primary keys	set null	EJB	
	invalid	EJB	
	wrong	EJB	\approx
Corrupt JNDI entries	set null	EJB	
	invalid	EJB	
	wrong	EJB	
Corrupt transaction method map	set null	EJB	
	invalid	EJB	
	wrong	EJB	\approx
Corrupt stateless session EJB attributes	set null	unnecessary	
	invalid	unnecessary	
	wrong	EJB+WAR	\approx
Corrupt data inside FastS	set null	WAR	
	invalid	WAR	
	wrong	WAR	\approx
Corrupt data inside SSM	corruption detected via checksum; bad object automatically discarded		
Corrupt data inside MySQL	database table repair needed		
Memory leak	intra-JVM	JVM/JBoss	
	outside application	OS kernel	
Bit flips in process memory		JVM/JBoss	\approx
Bit flips in process registers		JVM/JBoss	\approx
Bad system call return values		JVM/JBoss	

Table 2: Recovery from injected faults: worst case scenarios. Aside from EJB, JBoss, and operating system reboots, some faults require microrebooting eBid's Web component (WAR). In two cases no resuscitation is needed, because the injected fault is "naturally" expunged from the system after the first call fails. In the case of recovering persistent data, this is either done automatically (transaction rollback), or, in the case of injecting *wrong* data, manual reconstruction of the data in the DB is often required (indicated by \approx in the last column). We used the comparison-based fault detector for all experiments in this table.

end users do not experience any more failures after recovery. Figure 1 shows the results of one such experiment, in which we injected three different faults every 10 minutes. Session state is stored in FastS. We ran a load of 500 concurrent clients connected to one application server node; for our specific setup, this lead to a CPU load average of 0.7, which is similar to that seen in deployed Internet systems [29, 15]. Unless otherwise noted, we use 500 concurrent clients per node in each subsequent experiment.

Overall, using μ RBs instead of JVM restarts reduced the number of failed requests by 98%. Visually, the impact of a failure and recovery event can be estimated by the area of the corresponding dip in good T_{aw} , with larger dips indicating higher service disruption. The area of a T_{aw} dip is determined by its width (i.e., time to recover) and depth (i.e., the throughput of requests turned away during recovery). We now consider each factor in isolation.

Microreboots recover faster. The wider the dip in T_{aw} , the more requests arrive during recovery; since these requests fail, they cause the corresponding user actions to fail, thus retroactively marking the actions' requests as failed. We measured recovery time at various granularities and summarize the results in Table 3. In the two right columns we break down recovery time into how long the

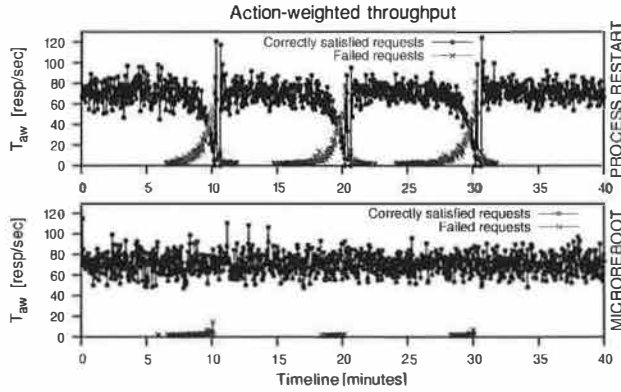


Figure 1: Using T_{aw} to compare JVM process restart to EJB microreboot. Each sample point represents the number of successful (failed) requests observed during the corresponding second. At $t=10$ min, we corrupt the transaction method map for *EntityGroup*, the EJB recovery group that takes the longest to recover. At $t=20$ min, we corrupt the JNDI entry for *RegisterNewUser*, the next-slowest in recovery. At $t=30$ min, we inject a transient exception in *BrowseCategories*, the entry point for all browsing (thus, the most-frequently called EJB in our workload). Overall, 11,752 requests (3,101 actions) failed when recovering with a process restart, shown in the top graph; 233 requests (34 actions) failed when recovering by microrebooting one or more EJBs. Thus, the average is 3,917 failed requests (1,034 actions) per process restart, and 78 failed requests (11 actions) per microreboot of one or more EJBs.

target takes to crash (be forcefully shut down) and how long it takes to reinitialize. EJBs recover an order of magnitude faster than JVM restart, which explains why the width of the good T_{aw} dip in the μRB case is negligible.

As described in Section 3, some EJBs have interdependencies, captured in deployment descriptors, that require them to be microrebooted together. *eBid* has one such recovery group, *EntityGroup*, containing 5 entity EJBs: *Category*, *Region*, *User*, *Item*, and *Bid* – any time one of these EJBs requires a μRB , we microreboot the entire *EntityGroup*. Restarting the entire *eBid* application is optimized to avoid restarting each individual EJB, which is why *eBid* takes less than the sum of all components to crash and start up. For the JVM crash, we use operating system-level `kill -9`.

All reboot-based recovery times are dominated by initialization. In the case of JVM-level restart, 56% of the time is spent initializing JBoss and its more than 70 services (transaction service takes 2 sec to initialize, embedded Web server 1.8 sec, JBoss’s control & management service takes 1.2 sec, etc.). Most of the remaining 44% startup time is spent deploying and initializing *eBid*’s EJBs and WAR. For each EJB, the deployer service verifies that the EJB object conforms to the EJB specification (e.g., has the required interfaces), then it allocates and initializes a container, sets up an object instance pool, sets up the security context, inserts an appropriate name-to-EJB mapping in JNDI, etc. Once initialization completes, the individual EJBs’ `start()` methods are invoked. Removing an EJB from the system follows a reverse path.

Microreboots reduce functional disruption during recovery. Figure 1 shows that good T_{aw} drops all the way to

Component name	μRB time (msec)	Crash (msec)	Reinit (msec)
AboutMe	551	9	542
Authenticate	491	12	479
BrowseCategories	411	11	400
BrowseRegions	416	15	401
BuyNow*	471	9	462
CommitBid	533	8	525
CommitBuyNow	471	9	462
CommitUserFeedback	531	9	522
DoBuyNow	427	10	417
EntityGroup*	825	36	789
IdentityManager*	461	10	451
LeaveUserFeedback	484	10	474
MakeBid	514	9	515
OldItem*	529	10	519
RegisterNewItem	447	13	434
RegisterNewUser	601	13	588
SearchItemsByCategory	442	14	428
SearchItemsByRegion	572	8	564
UserFeedback*	483	11	472
ViewBidHistory	507	11	496
ViewItemInfo	415	10	405
ViewItem	446	10	436
WAR (Web component)	1,028	71	957
Entire eBid application	7,699	33	7,666
JVM/JBoss process restart	19,083	≈ 0	$\approx 19,083$

Table 3: Average recovery times under load, in msec, for the individual components, the entire application, and the JVM/JBoss process. EJBs with a * superscript are entity EJBs, while the rest are stateless session EJBs. Averages are computed across 10 trials per component, on a single-node system under sustained load from 500 concurrent clients. Recovery for individual EJBs ranges from 411-601 msec.

zero during a JVM restart, i.e., the system serves no requests during that time. In the case of microrebooting, though, the system continues serving requests while the faulty component is being recovered. We illustrate this effect in Figure 2, graphing the availability of *eBid*’s functionality as perceived by the emulated clients. We group all *eBid* end user operations into 4 functional groups – *Bid/Buy/Sell*, *Browse/View*, *Search*, and *User Account* operations – and zoom in on one of the recovery events of Figure 1.

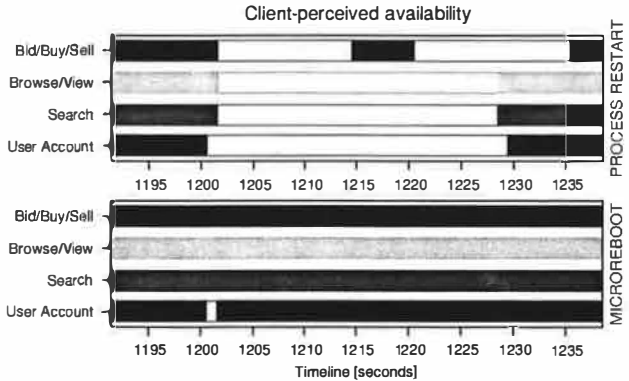


Figure 2: Functional disruption as perceived by end users. For each point t along the horizontal axis, a solid vertical line/bar indicates that, at time t , the service was *not* perceived as unavailable by *any* end user. A gap in an interval $[t_1, t_2]$ indicates that some request, whose processing spanned $[t_1, t_2]$ in time, eventually failed, suggesting the site was down.

While the faulty component is being recovered by microbooting, all operations in other functional groups succeed. Even within the “User Account” group itself, many operations are served successfully during recovery (however, since RegisterNewUser requests fail, we show the entire group as unavailable). Fractional service degradation compounds the benefits of swift recovery, further increasing end user-perceived availability of the service.

Microreboots reduce lost work. In Figure 1, a number of requests fail *after* JVM-level recovery has completed; this does not happen in the microreboot case. These failures are due to the session state having been lost during recovery (FastS does not survive JVM restarts). Had we used SSM instead of FastS, the JVM restart case would not have exhibited failed requests following recovery, and a fraction of the retroactively failed requests would have been successful, but the overall good T_{aw} would have been slightly lower (see Section 5.4). Using μ RBs in the FastS case allowed the system to both preserve session state across recovery and avoid cross-JVM access penalties.

5.3 Is Microbooting Useful in Clusters?

In a typical Internet cluster, the unit of recovery is a full node, which is small relative to the cluster as a whole. To learn whether μ RBs can yield any benefit in such systems, we built a cluster of 8 independent application server nodes. Clusters of 2-4 J2EE servers are typical in enterprise settings, with high-end financial and telecom applications running on 10-24 nodes [15]; a few gigantic services, like eBay’s online auction service, run on pools of clusters totaling 2000 application servers [11].

We distribute incoming load among nodes using a client-side load balancer *LB*. Under failure-free operation, *LB* distributes new incoming login requests evenly between the nodes and, for established sessions, *LB* implements session affinity (i.e., non-login requests are directed to the node on which the session was originally established). We inject a μ RB-recoverable fault from Table 2 in one of the server instances, say N_{bad} ; the failure detectors notice failures and report them to the recovery manager. When *RM* decides to perform a recovery, it first notifies *LB*, which redirects requests bound for N_{bad} uniformly to the good nodes; once N_{bad} has recovered, *RM* notifies *LB*, and requests are again distributed as before the failure.

Failover under normal load. We first explored the configuration that is most likely to be found in today’s systems: session state stored locally at each node; we use FastS. During failover, those requests that do not require session state, such as searching or browsing, will be successfully served by the good nodes; requests that require session state will fail. We injected a fault in the most-frequently called component (BrowseCategories) and ran the experiment in four clusters of different sizes; the load was 500 clients/node.

The left graph in Figure 3 shows the results. When recovering N_{bad} with a JVM restart, the number of user requests that fail is dominated by the number of sessions that

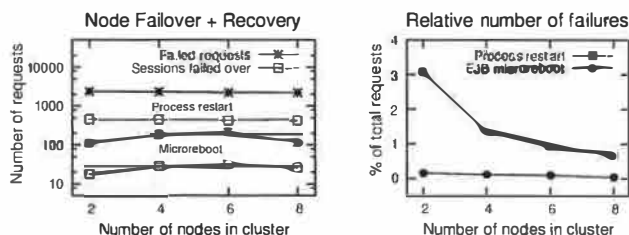


Figure 3: Failover under normal load. On the left we show the number of requests and failed-over sessions for the case of JVM restart and μ RB, respectively. On the right we show what fraction of total user requests failed in our test’s 10-minute interval, as a function of cluster size.

were established at the time of recovery on N_{bad} . In the case of EJB-level microbooting, the number of failed requests is roughly proportional to the number of requests that were in flight at the time of recovery or were submitted during recovery. Thus, as the cluster grows, the number of failed user requests stays fairly constant. When recovering with JVM restart, on average 2,280 requests failed; in the case of microbooting, 162 requests failed.

Although the relative benefit of microbooting decreases as the number of cluster nodes increases (right graph in Figure 3), recovering with a microreboot will always result in fewer failed requests than a JVM restart, regardless of cluster size or of how many clients each cluster node serves. Thus, it always improves availability. If a cluster aimed for the level of availability offered by today’s telephone switches, then it would have to offer six nines of availability, which roughly means it must satisfy 99.9999% of requests it receives (i.e., fail at most 0.0001% of them). Our 8-node cluster served 33.8×10^4 requests over the course of 10 minutes; extrapolated to a 24-node cluster of application servers, this implies 53.3×10^9 requests served in a year, of which a six-nines cluster can fail at most 53.3×10^3 . If using JVM restarts, this number allows for 23 single-node failovers during the whole year; if using microboots, 329 failures would be permissible.

We repeated some of the above experiments using SSM. The availability of session state during recovery was no longer a problem, but the per-node load increased during recovery, because the good nodes had to (temporarily) handle the N_{bad} -bound requests. In addition to the increased load, the session state caches had to be populated from SSM with the session state of N_{bad} -bound sessions. These factors resulted in an increased response time that often exceeded 8 sec when using JVM restarts; microbooting was sufficiently fast to make this effect unobservable. Overload situations are mitigated by overprovisioning the cluster, so we investigate below whether microbooting can reduce the need for additional hardware.

Microreboots preserve cluster load dynamics. We repeated the experiments described above using FastS, but doubled the concurrent user population to 1000 clients/node. The load spike we model is very modest compared to what can occur in production systems (e.g., on 9-11, CNN.com faced a 20-fold surge in load, which

caused their cluster to collapse under congestion [23]). We also allow the system to stabilize at the higher load prior to injecting faults (for this reason, the experiment's time interval was increased to 13 minutes). Since JVM restarts are more disruptive than microreboots, a mild two-fold change in load and stability in initial conditions favor full process restarts more than μ RBs, so the results shown here are conservative with respect to microrebooting. Figure 4 shows that response time was preserved while recovering with μ RBs, unlike when using JVM restarts.

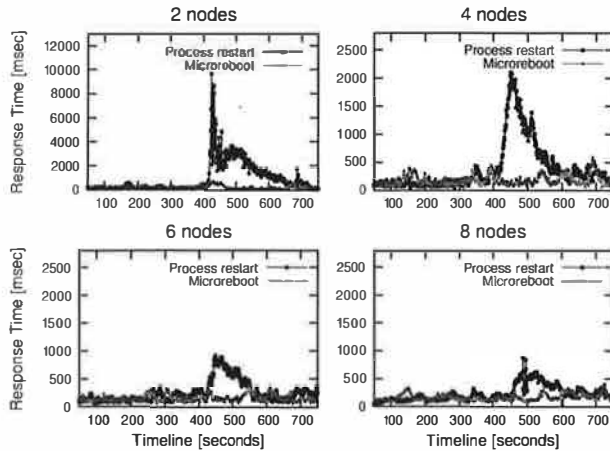


Figure 4: Failover under doubled load. We show average response time per request, computed over 1-second intervals, in 4 different cluster configurations (2,4,6,8 nodes). eBid uses FastS for storing session state, in both the JVM restart and microboot case. Vertical scales of the four graphs differ, to enhance visibility of details.

Stability of response time results in improved service to the end users. It is known that response times exceeding 8 seconds cause computer users to get distracted from the task they are pursuing and engage in others [31, 4], making this a common threshold for Web site abandonment [44]; not surprisingly, service level agreements at financial institutions often stipulate 8 seconds as a maximum acceptable response time [28]. We therefore measured how many requests exceeded this threshold during failover; Table 4 shows the corresponding results.

# of nodes	2	4	6	8
Process restart	3,227	530	55	9
EJB microboot	3	0	0	0

Table 4: Requests exceeding 8 sec during failover under doubled load.

We asked our colleagues in industry whether commercial application servers do admission control when overloaded, and were surprised to learn they currently do not [29, 15]. For this reason, cluster operators need to significantly overprovision their clusters and use complex load balancers, tuned by experts, in order to avert overload and oscillation problems. Microreboots reduce the need for overprovisioning or sophisticated load balancing. Since μ RBs are more successful at keeping response times below 8 seconds in our prototype, we would expect user experience to be improved in a clustered system that uses microboot-based recovery instead of process restarts.

5.4 Performance Impact

In this section we measure the performance impact our modifications have on steady-state fault-free throughput and latency. We measure the impact of our microreboot-enabling modifications on the application server, by comparing original JBoss 3.2.1 to the microreboot-enabled variant. We also measure the cost of externalizing session state into a remote state store by comparing eBid with FastS to eBid with SSM. Table 5 summarizes the results.

Configuration	Throughput [req/sec]	Average Latency [msec]
JBoss + eBidFastS	72.09	15.02
JBoss μ RB + eBidFastS	72.42	16.08
JBoss + eBidSSM	71.63	28.43
JBoss μ RB + eBidSSM	70.86	27.69

Table 5: Performance comparison: (a) original JBoss vs. microreboot-enabled JBoss μ RB; (b) intra-JVM session state storage (eBidFastS) vs. extra-JVM session state storage (eBidSSM).

Throughput varies less than 2% between the various configurations, which is within the margin of error. Latency, however, increases by 70-90% when using SSM, because moving state between JBoss and a remote session state store requires the session object to be marshalled, sent over the network, then unmarshalled; this consumes more CPU than if the object were kept inside the JVM. Since minimum human-perceptible delay is about 100 msec [31], we believe the increase in latency is of little consequence for an interactive Internet service like ours. Latency-critical applications can use FastS instead of SSM. The performance results are within the range of measurements done at a major Internet auction service, where latencies average 33-300 msec, depending on operation, and average throughput is 41 req/sec per node [16].

It is not meaningful to compare the performance of eBid to that of original RUBiS, because the semantics of the applications are different. For example, RUBiS requires users to provide a username and password *each time* they perform an operation requiring authentication. In eBid, users log in once at the beginning of their session; they are subsequently identified based on the HTTP cookies they supply to the server on every access. We refer the reader to [10] for a detailed comparison of performance and scalability for various architectures in J2EE applications.

6 A New Approach to Failure Management

The previous section showed microreboots to have significant quantitative benefits in terms of recovery time, functionality disruption, amount of lost work, and preservation of load dynamics in clusters. These quantitative improvements beget a qualitative change in the way we can manage failures in large-scale componentized systems; here we present some of these new possibilities.

6.1 Alternative Failover Schemes

In a microrebootable cluster, μ RB-based recovery should always be attempted first, prior to failover. As seen earlier, node failover can be destabilizing. In the first set of experiments in Section 5.3, failing requests over to good nodes while N_{bad} was recovering by μ RB resulted in 162 failed requests. In Figure 1, however, the average number of failures when requests continued being sent to the recovering node was 78. This shows that μ RB without failover improves user-perceived availability over failover and μ RB.

The benefit of pre-failover μ RB is due to the mismatch between node-level failover and component-level recovery. Coarse-grained failover prevents N_{bad} from serving a large fraction of the requests it could serve while recovering (Figure 2). Redirecting those requests to other nodes will cause many requests to fail (if not using SSM), or at best will unnecessarily overload the good nodes (if using SSM). Should the pre-failover μ RB prove ineffective, the load balancer can do failover and have N_{bad} rebooted; the cost of microbooting in a non- μ RB-curable case is negligible compared to the overall impact of recovery.

Using the average of 78 failed requests per microreboot instead of 162, we can update the computation for six-nines availability from Section 5.3. Thus, if using microreboots and *no failover*, a 24-node cluster could fail 683 times per year and still offer six nines of availability. We believe writing microrebootable software that is allowed to fail almost twice every day (683 times/year) is easier than writing software that is not allowed to fail more than once every 2 weeks (≈ 23 times/year for JVM restart recovery).

Another way to mitigate the coarseness of node-level failover is to use component-level failover; having reduced the cost of a reboot by making it finer-grain, *microfailover* seems a natural solution. Load balancers would have to be augmented with the ability to fail over only those requests that would touch the component(s) known to be recovering. There is no use in failing over any other requests. Microfailover accompanied by microreboot can reduce recovery-induced failures even further. Microfailover, however, requires the load balancer to have a thorough understanding of application dependencies, which might make it impractical for real Internet services.

6.2 User-Transparent Recovery

If recovery is sufficiently non-intrusive, then we can use low-level retry mechanisms to hide failure and recovery from callers – if it is brief, they won’t notice. Fortunately, the HTTP/1.1 specification [18] offers return code 503 for indicating that a Web server is temporarily unable to handle a request (typically due to overload or maintenance). This code is accompanied by a `Retry-After` header containing the time after which the Web client can retry.

We implemented call retry in our prototype. Previously, the first step in microbooting a component was the removal of its name binding from JNDI; instead, we bind

the component’s name to a sentinel during μ RB. If, while processing an idempotent request, a servlet encounters the sentinel on an EJB name lookup, the servlet container automatically replies with `[Retry-After 2 seconds]` to the client. We associated idempotency information with URL prefixes based on our understanding of eBid, but this could also be inferred using static call analysis. We measured the effect of HTTP/1.1 retry on calls to four different components, and found that transparent retry masked roughly half of the failures (Table 6); this corresponds to a two-fold increase in perceived availability.

Operation / component name	No retry	Retry	Delay & retry
ViewItem	23	16	8
BrowseCategories	20	8	0
SearchItemsByCategory	31	15	0
Authenticate	20	9	1

Table 6: Masking microreboots with HTTP/1.1 `Retry-After`. The data is averaged across 10 trials for each component shown.

The failed requests visible to end users were requests that had already entered the system when the microreboot started. To further reduce failures, we experimented with introducing a 200-msec delay between the sentinel rebind and beginning of the microreboot; this allowed some of the requests that were being processed by the about-to-be-microbooted component to complete. Of course, a component that has encountered a failure might not be able to process requests prior to recovery, unless only some instances of the EJB are faulty, while other instances are OK (a microreboot recycles all instances of that component). The last column in Table 6 shows a significant further reduction in failed requests. We did not analyze the tradeoff between number of saved requests and the 200-msec increase in recovery time.

6.3 Tolerating Lax Failure Detection

In general, downtime for an incident is the sum of the time to detect the failure (T_{det}), the time to diagnose the faulty component, and the time to recover. A failure monitor’s quality is generally characterized by how quick it is (i.e., T_{det}), how many of its detections are mistaken (false positive rate FP_{det}), and how many real failures it misses (false negative rate FN_{det}). Monitors make tradeoffs between these parameters; e.g., a longer T_{det} generally yields lower FP_{det} and FN_{det} , since more sample points can be gathered and their analysis can be more thorough.

Cheap recovery relaxes the task of failure detection in at least two ways. First, it allows for longer T_{det} , since the additional requests failing while detection is under way can be compensated for with the reduction in failed requests during recovery. Second, since false positives result in useless recovery leading to unnecessarily failing requests, cheaper recovery reduces the cost of a false positive, enabling systems to accommodate higher FP_{det} . Trading away some FP_{det} and T_{det} may result in a lower false negative rate, which could improve availability.

We illustrate T_{det} relaxation in the left graph of Figure 5. We inject a fault in the most frequently called EJB and delay recovery by T_{det} seconds, shown along the horizontal axis; we then perform recovery using either a JVM restart or a microreboot. The dotted line indicates that, with μRB -based recovery, a monitor can take up to 53.5 seconds to detect a failure, while still providing higher user-perceived availability than JVM restarts with immediate detection ($T_{\text{det}} = 0$). The two curves in the graph become asymptotically close for large values of T_{det} , because the number of requests that fail during detection (i.e., due to the delay in recovery) eventually dominate those that fail during recovery itself.

Doing real-time diagnosis instead of recovery has an opportunity cost. In this experiment, 102 requests failed during the first second of waiting; in contrast a microreboot averages 78 failed requests and takes 411-825 msec (Table 3), which suggests that microbooting *during* diagnosis would result in approximately the same number of failures, but offers the possibility of curing the failure before diagnosis completes.

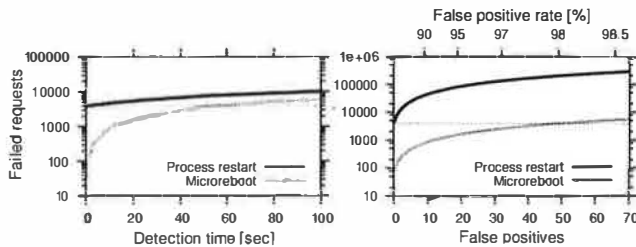


Figure 5: Relaxing failure detection with cheap recovery.

The right graph of Figure 5 shows the effect of false positives on end-user-perceived availability, given the averages from Figure 1: 3,917 failed requests per JVM restart, 78 requests per μRB . False positive detections occur inbetween correct positive detections; the false ones result in pointless recovery-induced downtime, while the correct ones lead to useful recovery. For simplicity, we assume $T_{\text{det}} = 0$. The graph plots the number of failed requests $f(n)$ caused by a sequence of n useless recoveries (triggered by false positives) followed by one useful recovery (in response to the correct positive). A given number n of false positives inbetween successive correct detections corresponds to a $FP_{\text{det}} = n/(n+1)$. The dotted line indicates that the availability achieved with JVM restarts and $FP_{\text{det}} = 0\%$ can be improved with μRB -based recovery even when false positive rates are as high as 98%.

Engineering failure detection that is both fast and accurate is difficult. Microreboots give failure detectors more headroom in terms of detection speed and false positives, allowing them to reduce false negative rates instead, and thus reduce the number of real failures they miss. Lower false negative rates can lead to higher availability. We would expect some of the extra headroom to also be used for improving the precision with which monitors pinpoint faulty components, since microbooting requires component-level precision, unlike JVM restarts.

6.4 Averting Failure with Microrejuvenation

Despite automatic garbage collection, resource leaks are a major problem for many large-scale Java applications; a recent study of IBM customers' J2EE e-business software revealed that production systems frequently crash because of memory leaks [33]. To avoid unpredictable leak-induced crashes, operators resort to preventive rebooting, or software rejuvenation [20]. Some of the largest U.S. financial companies reboot their J2EE servers daily [32] to recover memory, network sockets, file descriptors, etc. In this section we show that μRB -based rejuvenation, or *microrejuvenation*, can be as effective as a JVM restart in preventing leak-induced failures, but cheaper.

We wrote a server-side rejuvenation service that periodically checks the amount of memory available in the JVM; if it drops below M_{alarm} bytes, then the recovery service microreboots components in a rolling fashion until available memory exceeds a threshold $M_{\text{sufficient}}$; if all EJBs are microrebooted and $M_{\text{sufficient}}$ has not been reached, the whole JVM is restarted. Production systems could monitor a number of additional system parameters, such as number of file descriptors, CPU utilization, lock graphs for identifying deadlocks, etc.

The rejuvenation service does not have any knowledge of *which* components need to be microrebooted in order to reclaim memory. Thus, it builds a list of all components; as components are microrebooted, the service remembers how much memory was released by each one's μRB . The list is kept sorted in descending order by released memory and, the next time memory runs low, the rejuvenation service microrejuvenates components expected to release most memory, re-sorting the list as needed.

We induce memory leaks in two components: *ViewItem*, a stateless session EJB called frequently in our workload, and *Item*, an entity EJB part of the long-recovering *EntityGroup*. We choose leak rates that allow us to keep each experiment under 30 minutes.

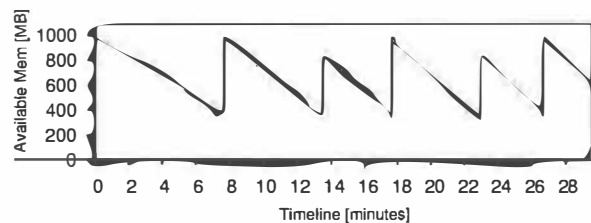


Figure 6: Available memory during microrejuvenation. We inject a 2 KB/invoction leak in *Item* and a 250 KB/invoction leak in *ViewItem*. M_{alarm} is set to 35% of the 1-GB heap (thus ≈ 350 MB) and $M_{\text{sufficient}}$ to 80% (≈ 800 MB).

In Figure 6 we show how free memory varies under a worst-case scenario for microrejuvenation: the initial list of components has the components leaking most memory at the very end. During the first round of microrejuvenation (interval [7.43-7.91] on the timeline), all of *eBid* ends up rebooted by pieces. During this time, *ViewItem* is found to have the most leaked memory, and

Item the second-most; the list of candidate components is reordered accordingly, improving the efficiency of subsequent rejuvenations. The second time M_{alarm} is reached, at $t = 13.8$, microbooting `ViewItem` is sufficient to bring available memory above threshold. On the third rejuvenation, both `ViewItem` and `Item` require rejuvenation; on the fourth, a `ViewItem` μRB is again sufficient; and so on.

Repeating the same experiment, but using whole rejuvenation via JVM restarts, resulted in a total of 11,915 requests failed during the 30-minute interval. When microrejuvenating with μRB s, only 1,383 requests failed – an order of magnitude improvement – and good T_{aw} never dropped to zero. The commonly used argument to motivate software rejuvenation is that it turns unplanned total downtime into planned total downtime; with microrejuvenation, we can further turn this planned total downtime into planned *partial* downtime.

7 Limitations of Recovery by Microreboot

It may appear that μRB s introduce three classes of problems: interruption of a component during a state update, improper reclamation of a microbooted component's external resources, and delay of a (needed) full reboot.

Impact on shared state. If state updates are atomic, as they are with databases, FastS, or SSM, there is no distinction between μRB s and process restarts from the state's perspective. However, the case of non-atomic updates to state shared between components is more challenging: microbooting one component may leave that state inconsistent, unbeknownst to the other components that share it. A JVM restart, on the other hand, reboots all components simultaneously, so it does not give them an opportunity to see the inconsistent state. J2EE best-practices documents discourage sharing state by passing references between components or using static variables, but we believe this should be a requirement enforced by a suitably modified JIT compiler. Alternatively, if the runtime detects unsafe state sharing practices, it should disable the use of μRB s for the application in question.

Not only does a JVM restart refresh all components, but it also discards the volatile shared state, regardless of whether it is inconsistent or not; μRB s allow that state to persist. In a crash-only system, state that survives the recovery of components resides in a state store that assumes responsibility for data consistency. In order to accomplish this, dedicated state repositories need APIs that are sufficiently high-level to allow the repository to repair the objects it manages, or at the very least to detect corruption. Otherwise, faults and inconsistencies perpetuate; this is why application-generic checkpoint-based recovery in Unix was found not to work well [27]. In the logical limit, all applications become stateless and recovery involves either microbooting the processing components, or repairing the data in state stores.

Interaction with external resources. If a component circumvents JBoss and acquires an external resource that the application server is not aware of, then microboot-

ing it may leak the resource in a way that a JVM/JBoss restart would not. For example, we experimentally verified that an EJB X can directly open a connection to a database without using JBoss's transaction service, acquire a database lock, then share that connection with another EJB Y . If X is microbooted prior to releasing the lock, Y 's reference will keep the database connection open even after X 's μRB , and thus X 's DB session stays alive. The database will not release the lock until after X 's DB session times out. In the case of a JVM restart, however, the resulting termination of the underlying TCP connection by the operating system would cause the immediate termination of the DB session and the release of the lock. If JBoss only knew X acquired a DB session, it could properly free the session even in the case of μRB .

While this example is contrived and violates J2EE programming practices, it illustrates the need for application components to obtain resources exclusively through the facilities provided by their platform.

Delaying a full reboot. The more state gets segregated out of the application, the less effective a reboot becomes at scrubbing this data. Moreover, our implementation of μRB does not scrub data maintained by the application server on behalf of the application, such as the database connection pool and various caches. Microboots also generally cannot recover from problems occurring at layers below the application, such as within the application server or the JVM; these require a full JVM restart instead.

When a full process restart is required, poor failure diagnosis may result in one or more ineffectual component-level μRB s. As discussed in Section 6.3, failure localization needs to be more precise for microboots than for JVM restarts. Using our recursive policy, microbooting progressively larger groups of components will eventually restart the JVM, but later than could have been done with better diagnosis. Even in this case, however, μRB s add only a small additional cost to the total recovery cost.

8 Generalizing beyond Our Prototype

Some J2EE applications are already microboot-friendly and require minimal changes to take advantage of our μRB -enabled application server. Based on our experience with other J2EE applications, we learned that the biggest challenges in making them 100% microbootable are (a) extricating session state handling from the application logic, and (b) ensuring that persistent state is updated with transactions. The rest is already done in our prototype server and can be leveraged across all J2EE applications.

While we feel J2EE makes it easier to write a microbootable application, because its model is amenable to state externalization and component isolation, we hope to see microboot support in other types of systems as well. In this section we describe design aspects that deserve consideration in such extensions.

Isolation: If there is one property of microbootable systems that is more critical than all the others, it is the partitioning of the system into fine-grain, well isolated

components. While such partitioning is a system-specific task, frameworks like J2EE and .NET [30] can help. Component isolation in J2EE is not enforced by lower-level (hardware) mechanisms, as would be the case with separate process address spaces; consequently, bugs in the Java virtual machine or the application server could result in state corruption crossing component boundaries. Depending on the system, stronger levels of isolation may be warranted, such as can be achieved with processes or virtual machines. Dependencies between components need to be minimized, because a dense dependency graph increases the size of recovery groups, making μ RBs take longer and be more disruptive.

Workload: Microreboots thrive on workloads consisting of fine-grain, independent requests; if a system is faced with long running operations, then individual components could be periodically microcheckpointed [42] to keep the cost of μ RBs low, keeping in mind the associated risk of persistent faults. In the same vein, requests need to be sufficiently self-contained, such that a fresh instance of a microbooted component can pick up a request and continue processing it where the previous instance left off.

Resources: Java does not offer explicit memory release or lease-based allocation, so the best we could do was to call the system garbage collector after μ RB. However, this form of resource reclamation does not complete in an amount of time that is independent of the size of the memory, unlike most traditional operating systems. We believe that efficient support for microreboots requires a nearly-constant-time resource reclamation mechanism, to allow microreboots to synchronously clean up resources.

9 Related Work

Our work has three major themes: reboot-based recovery, minimizing recovery time, and reducing disruption during recovery. In this section we discuss a small sample of work related to these themes.

Separation of control and data is key to reboot-based recovery. There are many ways to isolate subsystems (e.g., using processes, virtual machines [17], microkernels [25], protection domains [41], etc.). Isolated processing components appeared also in pre-J2EE transaction processing monitors, where each piece of system functionality (e.g., doing I/O with clients, writing to the transaction log) was a separate process communicating with the others using IPC or RPC. Session state was managed in memory by a dedicated component. Although the architecture did not scale very well, the “one component/one process” approach provided better isolation than monolithic architectures and would have been amenable to microbooting.

Baker [2] observed that emphasizing fast recovery over crash prevention has the potential to improve availability, and she described ways to build distributed file systems such that they recover quickly after crashes. In her design, a “recovery box” safeguards metadata in memory for recovery after a warm reboot. In our work, we provide components for a more general framework that both reduces

the impact of a crash and speeds up recovery.

Much work in Internet services has focused on reducing the functional disruption associated with recovering from a transient failure. Failover in clusters is the canonical example; Brewer [5] proposed the “DQ principle” as a way to understand how a partial failure in a multi-node service can be mapped to either a decrease in queries served per second, or a decrease in data returned per query.

Other research systems have embraced the approach of reducing downtime by recovering at sub-system levels. For example, Nooks [41] isolates drivers within lightweight protection domains inside the operating system kernel; when a driver fails, it can be restarted without affecting the rest of the kernel. Farsite [1], a peer-to-peer file system, has been recently restructured as a collection of crash-only components, that are recovered through rebooting. These systems provide examples of microbootable systems and lend credibility to the belief that non-J2EE systems can be structured for effective microrebootability.

10 Conclusions

Employing reboot-based recovery does not mean that the root causes of failures should not be identified and fixed. Rebooting simply provides a separation of concerns between diagnosis and recovery, consistent with the observation that the former is not always a prerequisite for the latter. Moreover, attempting to recover a reboot-curable failure by anything other than a reboot entails the risk of taking longer and being more disruptive than a reboot would have been in the first place, thus hurting availability.

By completely separating process recovery from data recovery, and delegating the latter to specialized state stores, we enabled the use of microreboots to achieve process recovery. In our experiments, microreboots cured the majority of failures that were empirically observed to cause downtime in deployed Internet services. Compared to process restart-based recovery, microbooting is an order of magnitude faster and less disruptive, even in multi-node clusters.

Regardless of fault, in microbootable systems one should first attempt microboot-based recovery: it does not take long and costs very little. Skipping node failover in clusters and microbooting the faulty node can improve availability over the commonly-used “fail over and reboot node” approach. Microboot-based recovery can achieve higher levels of availability even when false positive rates in fault detection are as high as 98%. Using microreboots, we were able to reclaim memory leaks in our prototype application without shutting it down, improving availability by an order of magnitude.

There is a significant limitation in developing bug-free software beyond a certain size. Accepting bugs as a fact, we argue that structuring systems for cheap reboot-based recovery provides a promising path toward dependable large-scale software.

Acknowledgments

We would like to thank David Cheriton and our colleagues in the Recovery-Oriented Computing project for early feedback on this work. We are indebted to our shepherd Jason Nieh, the anonymous OSDI reviewers, and Katerina Argyraki, Kim Keeton, Adam Messinger, Martin Rinard, and Westley Weimer for patiently helping us improve this paper.

References

- [1] A. Adya, W. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, 2002.
- [2] M. Baker and M. Sullivan. The Recovery Box: Using fast recovery to provide high availability in the UNIX environment. In *Proc. Summer USENIX Technical Conference*, San Antonio, TX, 1992.
- [3] M. Barnes. J2EE application servers: Market overview. The Meta Group, March 2004.
- [4] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating user-perceived quality into web server design. In *Proc. 9th International WWW Conference*, Amsterdam, Holland, 2000.
- [5] E. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, July 2001.
- [6] P. A. Broadwell, N. Sastry, and J. Traupman. FIG: A prototype tool for online verification of recovery mechanisms. In *Workshop on Self-Healing, Adaptive and Self-Managed Systems*, New York, NY, 2002.
- [7] K. Buchacker and V. Sieh. Framework for testing the fault-tolerance of systems including OS and network aspects. In *Proc. IEEE High-Assurance System Engineering Symposium*, Boca Raton, FL, 2001.
- [8] G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proc. 8th Workshop on Hot Topics in Operating Systems*, Elmau, Germany, 2001.
- [9] G. Candea and A. Fox. Crash-only software. In *Proc. 9th Workshop on Hot Topics in Operating Systems*, Lihue, Hawaii, 2003.
- [10] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of EJB applications. In *Proc. 17th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Seattle, WA, 2002.
- [11] M. Chen, A. Zheng, J. Lloyd, M. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *Proc. Intl. Conference on Autonomic Computing*, New York, NY, 2004.
- [12] T. C. Chou. Beyond fault tolerance. *IEEE Computer*, 30(4):31–36, 1997.
- [13] T. C. Chou. Personal communication. Oracle Corp., 2003.
- [14] H. Cohen and K. Jacobs. Personal comm. Oracle, 2002.
- [15] S. Duvur. Personal comm. Sun Microsystems, 2004.
- [16] Information obtained under an agreement that prohibits disclosure of the company's name, May 2004.
- [17] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proc. 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, 2003.
- [18] J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. Internet RFC 2616, June 1999.
- [19] J. Gray. Why do computers stop and what can be done about it? In *Proc. 5th Symp. on Reliability in Distributed Software and Database Systems*, Los Angeles, CA, 1986.
- [20] Y. Huang, C. M. R. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In *Proc. 25th International Symposium on Fault-Tolerant Computing*, Pasadena, CA, 1995.
- [21] JBoss web page. <http://www.jboss.org/>.
- [22] Keynote Systems. <http://www.keynote.com/>.
- [23] W. LeFebvre. CNN.com—Facing a world crisis. Talk at *15th USENIX Systems Administration Conference*, 2001.
- [24] H. Levine. Personal communication. EBates.com, 2003.
- [25] J. Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70–77, 1996.
- [26] B. Ling, E. Kiciman, and A. Fox. Session state: Beyond soft state. In *Proc. 1st Symposium on Networked Systems Design and Implementation*, San Francisco, CA, 2004.
- [27] D. E. Lowell, S. Chandra, and P. M. Chen. Exploring failure transparency and the limits of generic recovery. In *Proc. 4th Symposium on Operating Systems Design and Implementation*, San Diego, CA, 2000.
- [28] G. Messer. Personal communication. US Bancorp, 2004.
- [29] A. Messinger. Personal comm. BEA Systems, 2004.
- [30] Microsoft. *The Microsoft .NET Framework*. Microsoft Press, Redmond, WA, 2001.
- [31] R. Miller. Response time in man-computer conversational transactions. In *Proc. AFIPS Fall Joint Computer Conference*, volume 33, 1968.
- [32] N. Mitchell. IBM Research. Personal Comm., 2004.
- [33] N. Mitchell and G. Sevitsky. LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *Proc. 17th European Conf. on Object-Oriented Programming*, Darmstadt, Germany, 2003.
- [34] B. Murphy and T. Gent. Measuring system and software reliability using an automated data collection process. *Quality and Reliability Engineering Intl.*, 11:341–353, 1995.
- [35] A. Pal. Personal communication. Yahoo!, Inc., 2002.
- [36] D. Reimer. IBM Research. Personal comm., 2004.
- [37] RUBiS project web page. <http://rubis.objectweb.org/>.
- [38] W. D. Smith. TPC-W: Benchmarking an E-Commerce solution. Transaction Processing Council, 2002.
- [39] M. Sullivan and R. Chillarege. Software defects and their impact on system availability – a study of field failures in operating systems. In *Proc. 21st International Symposium on Fault-Tolerant Computing*, Montréal, Canada, 1991.
- [40] Sun Microsystems. <http://java.sun.com/j2ee/>.
- [41] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *Proc. 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, 2003.
- [42] K. Whisnant, Z. Kalbarczyk, and R. Iyer. Micro-checkpointing: Checkpointing for multithreaded applications. In *Proc. IEEE Intl. On-Line Testing Workshop*, 2000.
- [43] A. P. Wood. Software reliability from the customer view. *IEEE Computer*, 36(8):37–42, Aug. 2003.
- [44] Zona research bulletin: The need for speed II, Apr. 2001.

Automated Worm Fingerprinting

Sumeet Singh, Cristian Estan, George Varghese and Stefan Savage
Department of Computer Science and Engineering
University of California, San Diego

Abstract

Network worms are a clear and growing threat to the security of today's Internet-connected hosts and networks. The combination of the Internet's unrestricted connectivity and widespread software homogeneity allows network pathogens to exploit tremendous parallelism in their propagation. In fact, modern worms can spread so quickly, and so widely, that no human-mediated reaction can hope to contain an outbreak.

In this paper, we propose an automated approach for quickly detecting previously unknown worms and viruses based on two key behavioral characteristics – a common exploit sequence together with a range of unique sources generating infections and destinations being targeted. More importantly, our approach – called “content sifting” – automatically generates *precise* signatures that can then be used to filter or moderate the spread of the worm *elsewhere* in the network.

Using a combination of existing and novel algorithms we have developed a scalable content sifting implementation with low memory and CPU requirements. Over months of active use at UCSD, our *Earlybird* prototype system has automatically detected and generated signatures for all pathogens known to be active on our network as well as for several *new* worms and viruses which were *unknown* at the time our system identified them. Our initial experience suggests that, for a wide range of network pathogens, it may be practical to construct fully automated defenses – even against so-called “zero-day” epidemics.

1 Introduction

In the last three years, large-scale Internet worm outbreaks have *profoundly* demonstrated the threat posed by self-propagating programs. The combination of widespread software homogeneity and the Internet's unrestricted communication model creates an ideal climate for infectious pathogens. Worse, each new epidemic has demonstrated increased speed, virulence or sophistication over its predecessors. While the Code Red worm took over fourteen hours to infect its vulnerable population in 2001, the Slammer worm, released some 18 months later, did the same in under 10 minutes [22, 21]. The Code Red worm is thought to have infected roughly 360,000 hosts, while, by some estimates, the Nimda worm compromised over two million [8]. While early worms typically spread by a single mechanism and did little else, modern variants such as SoBig.F and My-

Doom can spread through multiple vectors and have added backdoors, mail-relays and denial-of-service attacks to their payloads.

Unfortunately, our current ability to defend against these outbreaks is extremely poor and has not advanced significantly since the Code Red episode in mid-2001. In fact, the basic approach of detection, characterization, and containment has not changed significantly over the last five years. Typically, new worms are detected in an ad hoc fashion by a combination of intrusion detection systems and administrator legwork. Then, after isolating an instance of the worm, skilled security professionals manually characterize a worm *signature* and finally, this signature is used to contain subsequent infections via updates to anti-virus software and network filtering products. While this approach is qualitatively sound, it is quantitatively insufficient. Manual signature extraction is an expensive, slow, manual procedure that can take hours or even days to complete. It requires isolating a new worm, decompiling it, looking for invariant code sequences and testing the signature for uniqueness. However, recent simulations by Moore et al. suggest that an effective worm containment can require a reaction time of well under sixty seconds [23]. More concretely, consider that in the time it took to read this section, the Slammer worm had contacted well over a **billion** distinct Internet hosts.

This paper investigates the challenges in addressing this problem and describes a prototype system, called *Earlybird*, that can automatically detect and contain new worms on the network using precise signatures. Our approach, which we call *content sifting*, is based on two observations: first, that some portion of the content in existing worms is invariant – typically the code exploiting a latent host vulnerability – and second, that the spreading dynamics of a worm is atypical of Internet applications. Simply stated, it is rare to observe the same string recurring within packets sent from many sources to many destinations. By sifting through network traffic for content strings that are both frequently repeated and widely dispersed, we can automatically identify new worms and their precise signatures.

In our prototype system, we have developed approximate versions of this algorithm that are amenable to high-speed implementation. In live experiments on a portion of the UCSD campus network, we have deployed *Earlybird* and demonstrated that it can automatically extract the signature for all known active worms (e.g. CodeRed,

Slammer). Moreover, during our experiments Earlybird detected and extracted a signature for the Blaster, MyDoom and Kibuv.B worms – significantly before they had been publicly disclosed and hours or days before any public detection signatures were distributed. Finally, in our testing over a period of eight months, we have experienced relatively few false positives and exceptions are typically due to structural properties of a few popular protocols (SPAM via SMTP and NetBIOS) that recur consistently and can be procedurally “white-listed”.

The remainder of this paper is structured as follows. In Section 2 we survey the field of worm research that we have built upon and describe how it motivates our work. Section 3 describes how we define worm behavior. Section 4 outlines a naive approach to detecting such behaviors, followed by a concrete description of practical content sifting algorithms in Section 5. Section 6 describes the implementation of the Earlybird prototype and an analysis of our live experiments using it. We describe limitations and extensions in Section 7. Finally, in Section 8 we summarize our findings and conclude.

2 Background and Related Work

Worms are simply small programs. They spread by exploiting a latent software vulnerability in some popular network service – such as email, Web or terminal access – seizing control of program execution and then sending a copy of themselves to other susceptible hosts.

While the potential threat posed by network worms has a long past – originating with fictional accounts in Gerrold’s “When Harlie was One” and Brunner’s “Shockwave Rider” – it is only recently that this threat has enjoyed significant research attention. Fred Cohen first lay the theoretical foundations for understanding computer viruses in 1984 [4, 5], and the Internet worm of 1988 demonstrated that self-replication via a network could dramatically amplify the virulence of such pathogens [33, 39]. However, the analysis and understanding of network worms did not advance substantially until the CodeRed outbreak of 2001. In this section, we attempt to summarize the contemporary research literature – especially in its relation to our own work.

The first research papers in the “modern worm era” focused on characterizations and analyses of particular worm outbreaks. For example, Moore et al. published one of the first empirical analyses of the CodeRed worm’s growth, based on unsolicited scans passively observed on an unused network [22]. Further, the authors estimated the operational “repair” rate by actively probing a subsample of the 360,000 infected sites over time. They found that, despite unprecedented media coverage, the repair rate during the initial outbreak averaged under 2 percent per day. This reinforces our belief that fully automated intervention is necessary to effectively man-

age worm outbreaks. Staniford et al.’s landmark paper anticipated the development of far faster worms and extrapolated their growth analytically [42] – foreshadowing the release of the Slammer worm in 2002. Moore et al. subsequently analyzed the Slammer outbreak and estimated that almost *all* of the Internet address space was scanned by the worm in under 10 minutes – limited only by bandwidth constraints at the infected sites [21]. This experience also motivates the need for fast and automated reaction times. Finally, based on these results, Moore et al. analyzed the engineering requirements for reactive defenses – exploring the tradeoffs between reaction time, deployment and the granularity of containment mechanisms (signature based vs. IP address based) [23]. Two of their key findings motivate our work.

First, they demonstrated that signature-based methods can be an order of magnitude more effective than simply quarantining infected hosts piecemeal. The rough intuition for this is simple: if a worm can compromise a new host with an average latency of x seconds, then an address based quarantine can must react more quickly than x seconds to prevent the worm from spreading. By contrast, a signature based system can, in principle, halt all subsequent spreading once a signature is identified. The second important result was their derivation, via simulation, of “benchmarks” for how quickly such signatures must be generated to offer effective containment. Slow-spreading worms, such as CodeRed can be effectively contained if signatures are generated within 60 minutes, while containing high-speed worms, such as Slammer, may require signature generation in well under 5 minutes – perhaps as little as 60 seconds. Our principal contribution is demonstrating practical mechanisms for achieving this requirement.

In the remainder of this section we examine existing techniques for detecting worm outbreaks, characterizing worms and proposed countermeasures for mitigating worm spread.

2.1 Worm Detection

Three current classes of methods are used for detecting new worms: scan detection, honeypots, and behavioral techniques at end hosts. We consider each of these in turn.

Worms spread by selecting susceptible target hosts, infecting them over the network, and then repeating this process in a distributed recursive fashion. Many existing worms, excepting email viruses, will select targets using a random process. For instance, CodeRed selected target IP addresses uniformly from the entire address space. As a result, a worm may will be highly unusual in the number, frequency and distribution of addresses that it scans. This can be leveraged to detect worms in several ways.

To monitor random scanning worms from a global per-

spective, one approach is to use *network telescopes* – passive network monitors that observe large ranges of unused, yet routable, address space [25, 22, 26]. Under the assumption that worms will select target victims at random, a new worm will scan a given network telescope with a probability directly proportional to the worm’s scan rate and the network telescope’s “size”; that is, the number of IP addresses monitored. Consequently, large network telescopes will be able to detect fast spreading worms of this type fairly quickly. At the enterprise level, Stanford provides a comprehensive analysis of the factors impacting the ability of a network monitor to successfully detect and quarantine infected hosts in an on-line fashion [41].

However, there are two key limitations to the scan detection approach. First, it is not well suited to worms which spread in a non-random fashion, such as e-mail viruses or worms spread via instant messenger or peer-to-peer applications. Such worms generate a target list from address books or buddy lists at the victim and therefore spread *topologically* – according to the implicit relationship graph between individuals. Consequently, they do not exhibit anomalous scanning patterns and will not be detected as a consequence. The second drawback is that scan detection can only provide the IP address of infected sites, not a signature identifying their behavior. Consequently, defenses based on scan detection must be an order of magnitude faster than those based on signature extraction [23].

A different approach to worm detection is demonstrated by *Honeypots*. First introduced to the community via Cliff Stoll’s book, “The Cuckoo’s Egg”, and Bill Cheswick’s paper “An Evening with Berferd”, honeypots are simply monitored idle hosts with untreated vulnerabilities. Any outside interaction with the host is, by definition, unsolicited and any malicious actions can be observed directly. Consequently, any unsolicited outbound traffic generated by a honeypot represents undeniable evidence of an intrusion and possibly a worm infection. Moreover, since the honeypot host is directly controlled, malicious code can be differentiated from the default configuration. In this manner, the “body” of a worm can be isolated and then analyzed to extract a signature. This approach is commonly used for acquiring worm instances for manual analysis [18]. There are two principal drawbacks to honeypots: they require a significant amount of slow manual analysis and they depend on the honeypot being quickly infected by a new worm.

Finally, a technique that has found increasing traction in the commercial world (e.g. via recently acquired startups, Okena and Entracept) is host-based behavioral detection. Such systems dynamically analyze the patterns of system calls for anomalous activity [31, 28, 3] indicating code injection or propagation. For example, attempts

to send a packet from the same buffer containing a received packet is often indicative of suspicious activity. While behavioral techniques are able to leverage large amounts of detailed context about application and system behavior, they can be expensive to manage and deploy ubiquitously. Moreover, end-host systems can, by definition, only detect an attack against a single host and not infer the presence of a large-scale outbreak. Clearly, from a management, cost and reuse standpoint, it is ideal to detect and block new attacks in the network. That said, end-host approaches offer a level of sensitivity that is difficult to match in the network and can be a useful complement – particularly for detecting potential slow or stealthy worms that do not leave a significant imprint on the network.

2.2 Characterization

Characterization is the process of analyzing and identifying a new worm or exploit, so that targeted defenses may be deployed.

One approach is to create a priori *vulnerability signatures* that match known exploitable vulnerabilities in deployed software [44, 45]. For example, a vulnerability signature for the Slammer worm might match all UDP traffic on port 1434 that is longer than 100 bytes. By searching for such traffic, either in the network or on the host, a new worm exploiting the same vulnerability will be revealed. This is very similar to traditional intrusion detection systems (IDS), such as Snort [1] and Bro [29], which compare traffic content to databases of strings used in known attacks. This general approach has the advantage that it can be deployed before the outbreak of a new worm and therefore can offer an added measure of defense. However, this sort of proactive characterization can only be applied to vulnerabilities that are already well-known and *well-characterized* manually. Further, the tradeoff between vulnerability signature specificity, complexity and false positives remains an open question. Wang et al’s Shield, is by far the best-known vulnerability blocking system and it focuses on an end-host implementation precisely to better manage some of these tradeoffs [44]. We do not consider this approach further in this paper, but we believe it can be a valuable complement to the automated signature extraction alternative we explore.

The earliest automation for signature extraction is due to Kephart and Arnold [15]. Their system, used commercially by IBM, allows viruses to infect known “decoy” programs in a controlled environment, extracts the infected (i.e., modified) regions of the decoys and then uses a variety of heuristics to identify invariant code strings across infection instances. Among this set of candidates an “optimal” signature is determined by estimating the false positive probability against a measured corpus of

n-grams found in normal computer programs. This approach is extremely powerful, but assumes the presence of a known instance of a virus and a controlled environment to monitor.

The former limitation is partially addressed by the Honeycomb system of Kreibich and Crowcroft [17]. Honeycomb is a host-based intrusion detection system that automatically generates signatures by looking for longest common subsequences among sets of strings found in message exchanges. This basic procedure is similar to our own, but there are also important structural and algorithmic differences between our two approaches, the most important of which is scale. Honeycomb is designed for a host-based context with orders of magnitude less processing required. To put this in context, our Earlybird system currently processes more traffic in one second than the prototype Honeycomb observed in 24 hours. However, one clear advantage offered by the host context is its natural imperviousness to network evasion techniques [30]. We discuss this issue further in Section 7.

Finally, over the last two years of Earlybird's development [34, 35, 37], the clearest parallels can be drawn to Kim and Karp's contemporaneously-developed Autograph system [16]. Like Earlybird, Autograph also uses network-level data to infer worm signatures and both systems employ Rabin fingerprints to index counters of content substrings and use white-lists to set aside well-known false positives. However, there are several important differences as well. First, Autograph relies on a prefiltering step that identifies flows with suspicious scanning activity (particularly the number of unsuccessful TCP connection attempts) before calculating content prevalence. By contrast, Earlybird measures the prevalence of *all* content entering the network and only then considers the addressing activity. This difference means that Autograph cannot detect large classes of worms that Earlybird can – including almost all e-mail borne worms, such as MyDoom, UDP-based worms such as Slammer, spoofed source worms, or worms carried via IM or P2P clients. Second, Autograph has extensive support for distributed deployments – involving active cooperation between multiple sensors. By contrast, Earlybird has focused almost entirely on the algorithmics required to support a robust and scalable wire-speed implementation in a *single* sensor and only supports distribution through a centralized aggregator. Third, Earlybird is an on-line system that has been in near-production use for eight months and handles over 200 megabits of live traffic, while, as described, Autograph is an off-line system that has only been evaluated using traces. Finally, there are many differences in the details of the algorithms used (e.g. Autograph breaks content into non-overlapping variable-length chunks while Earlybird manages over-

lapping fixed-length content strings over each byte offset) although it is not currently clear what the impact of these differences is.

2.3 Containment

Containment refers to the mechanism used to slow or stop the spread of an active worm. There are three containment mechanisms in use today: host quarantine, string-matching and connection throttling. Host quarantine is simply the act of preventing an infected host from communicating with other hosts – typically implemented via IP-level access control lists on routers or firewalls. String-matching containment – typified by signature-based network intrusion prevention systems (NIPS) – matches network traffic against particular strings, or signatures, of known worms and can then drop associated packets. To enable high-bandwidth deployments, several hardware vendors are now producing high-speed string matching and regular expression checking chips for worm and virus filtering. Lockwood et al. describe an FPGA-based research prototype programmed for this application [19]. Finally, a different strategy, proposed by Twycross and Williamson [43], is to proactively limit the rate of all outgoing connections made by a machine and thereby slow – but not stop – the spread of any worm. Their approach was proposed in a host context, but there is no reason such connection throttling cannot be applied at the network level as well.

In this paper, we assume the availability of string-matching containment (perhaps in concert with throttling) and our Earlybird prototype generates signatures for a Snort in-line intrusion detection system – blocking all packets containing discovered worm signatures.

3 Defining Worm Behavior

Network worms, due to their distinct purpose, tend to behave quite differently from the popular client-server and peer-to-peer applications deployed on today's networks. In this section we explore these key behaviors in more detail and how they can be exploited to detect and characterize network worms.

3.1 Content invariance

In all existing worms of which we are aware, some or all of the worm program is invariant across every copy. Typically, the entire worm program is identical across every host it infects. However, some worms make use of limited polymorphism – by encrypting each worm instance independently and/or randomizing filler text. In these cases, much of the worm body is variable, but key portions are still invariant (e.g., the decryption routine). For the purposes of this paper, we assume that a worm has some amount of invariant content or has relatively few variants. We discuss violations of this assumption in Section 7.

3.2 Content prevalence

Since worms are designed foremost to *spread*, the invariant portion of a worm's content will appear frequently on the network as it spreads or attempts to spread. Conversely, content which is not prevalent will not represent the invariant portion of a worm and therefore is not a useful candidate for constructing signatures.

3.3 Address dispersion

For the same reasons, the number of distinct hosts infected by a worm will grow over time. Consequently, packets containing a live worm will tend to reflect a variety of different source and destination addresses. Moreover, during a major outbreak, the number of such addresses can grow extremely quickly. Finally, it is reasonable to expect that the distribution of these addresses will be far more uniform than typical network traffic which can have significant clustering [9].¹ In this paper we only take advantage of the first of these three observations, but we believe there is potential value in considering all of them.

4 Finding worm signatures

From these assumptions, we can conclude that network worms must generate significant traffic to spread and that this traffic will contain common substrings and will be directed between a variety of different sources and destinations. While it is not yet clear that this characterization is *exclusively* caused by worms, for now we will assume that identifying this traffic pattern is sufficient for detecting worms. We examine the issue of false positives later in the paper. In principle, detecting this traffic pattern is relatively straightforward. Figure 1 shows an idealized algorithm that achieves this goal. For each network packet, the content is extracted and all substrings processed. Each substring is indexed into a prevalence table that increments a count field for a given substring each time it is found. In effect, this table implements a histogram of all observed substrings. To maintain a count of unique source and destination addresses, each table entry also maintains two lists, containing IP addresses, that are searched and potentially updated each time a substring count is incremented. Sorting this table on the substring count and the size of the address lists will produce the set of likely worm traffic. Better still, the table entries meeting this worm behavior criteria are exactly those containing the invariant substrings of the worm. *It is these substrings that can be used as signatures to filter the worm out of legitimate network traffic.*

We call this approach *content sifting* because it effectively implements a high-pass filter on the contents of network traffic. Network content which is not prevalent

¹As described earlier, the presence of "dark" IP addresses can also provide qualitatively strong evidence of worm-like behavior.

```
ProcessTraffic(payload,srcIP,dstIP)
1  prevalence[payload]++
2  Insert(srcIP,dispersion[payload].sources)
3  Insert(dstIP,dispersion[payload].dests)
4  if (prevalence[payload]>PrevalenceTh
5     and size(dispersion[payload].sources)>SrcDispTh
6     and size(dispersion[payload].dests)>DstDispTh)
7     if (payload in knownSignatures)
8         return
9     endif
10    Insert(payload,knownSignatures)
11    NewSignatureAlarm(payload)
12 endif
```

Figure 1: The idealized content sifting algorithm detects all packet contents that are seen often enough and are coming from enough sources and going to enough destinations. The value of the detection thresholds and the time window over which each table is used are both parameters of the algorithm.

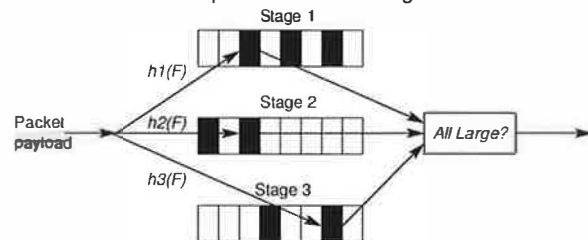


Figure 2: *Multi-stage Filters*. A piece of content is hashed using hash function $h1$ into a Stage 1 table, $h2$ into a Stage 2 table, etc and each table entry contains a counter that is incremented. If *all* the hashed counters are above the prevalence threshold, then the content string is saved for address dispersion measurements. In previous work we have shown that the probability of an approximation error decreases exponentially with the number of stages and consequently is extremely small in practice [10].

or not widely dispersed is sifted out, leaving only the worm-like content. However, while content sifting can correctly identify worm signatures, the basic algorithm we have described is far too inefficient to be practical. In the next section we describe algorithms for approximating correctness in exchange for efficiency and practicality.

5 Practical Content Sifting

For automated signature extraction to scale to high-speed links, the algorithms must have small processing requirements (ideally well-suited to parallelization), and small memory requirements. Finally, to allow arbitrary deployment strategies, the algorithm should not *depend* on having a symmetric vantage point in the network. To satisfy these requirements, we now describe scalable and accurate algorithms for estimating content prevalence and address dispersion, and techniques for managing CPU overload through smooth tradeoffs between detection time and overhead. For simplicity, in this section we describe our algorithms at packet (and not flow) granularity.

5.1 Estimating content prevalence

Identifying common content involves finding the packet payloads that appear at least x times among the N packets sent during a given interval. However, a table indexed

by payload can quickly consume huge amounts of memory. For example, on a fully loaded 1 Gbps link, this naive approach could generate a 1 GByte table in less than 10 seconds. Memory consumption can be reduced considerably by indexing the table using a fixed size hash of the packet payload instead of the full payload. After a certain hash value has repeated $x - 1$ times, the next packet with this hash is reported. In the absence of collisions, the associated content will have appeared exactly x times. By selecting a hash function with suitably large range (e.g., 32 or 64 bits) the collision probability can be minimized. Assuming a 16 byte hash table entry and an average packet size of 500 bytes, this algorithm would take over 4 minutes to generate the same 1 GByte table.

Memory efficiency can be improved further by observing that identifying prevalent content is isomorphic to the well-studied problem of identifying high-bandwidth flows, frequently called “heavy hitters” [13, 10]. By modifying the definition of “flow” to reflect content fingerprints instead of the *(srcip, dstip, srcport, dstport, protocol)* tuple used for flow analysis, heavy-hitter approximation algorithms can be used to find prevalent content using comparatively small amounts of memory.

Our prototype uses *multi-stage filters* with conservative update to dramatically reduce the memory footprint of the problem (see Figure 2 for a general description and [13, 10] for a thorough analysis). While simple, we believe this notion of using a content signature as a “flow identifier” on which to maintain counters is a powerful technique.²

An important modification is to append the destination port and protocol to the content before hashing. Since worms typically target a particular service (they are designed to exploit a vulnerability in that service) this will not impact the ability to track worm traffic, but can effectively exclude large amounts of prevalent content not generated by worms (i.e., potential false positives).³ For example, if two users on the same network both download the Yahoo home page they will receive many packets with identical payloads. However, traffic sent from the Web server will be directed to a so-called “ephemeral” port selected by each client. Since these ports are selected independently, adding them to the hash input will generally differentiate these different clients even when the content being carried is identical.

So far, we have only discussed content at the *whole*

²We are not the first to use hashing techniques to analyze the content makeup of network traffic. Snoeren et al. and Duffield et al. both use hashing to match packet observations across a network [38, 7], and both Spring et al. and Muthitacharoen et al. use Rabin fingerprints for compressing content sent over a network [40, 27].

³Note that it is possible for this assumption to be violated under unusual circumstances. In particular, the Witty worm exploited promiscuous network devices and only required a fixed *source* port to exploit its vulnerability – the destination port was random [6]. Catching this worm required us to maintain an additional matching table in which the source port is appended to the hash output instead.

packet granularity. While this is sufficient for detecting most existing worms, it is easy to envision worms for which the invariant content is a string smaller than a single packet or for which the invariant content occurs at different offsets in each instance. However, detecting common strings of at least a minimum length is computationally complex. Instead we address the related – yet far easier – problem of detecting repeating strings with a small fixed length β . As with full packet contents, storing individual substrings can require exorbitant memory and computational resources. Instead, we use a method similar to the one proposed by Manber for finding similar files in a large file system [20]. We compute a variant of Rabin fingerprints for all possible substrings of a certain length [32]. As these fingerprints are polynomials they can be computed incrementally while retaining the property that two equal substrings will generate the same fingerprint, no matter where they are in a packet.

However, each packet with a payload of s bytes has $s - \beta + 1$ strings of length β , so the memory references used per packet is still substantially greater than that consumed by a single per-packet hash. In Section 5.3, we describe a technique called *value sampling* to considerably reduce memory references.

5.2 Estimating address dispersion

While content prevalence is the key metric for identifying potential worm signatures, address dispersion is critical for avoiding false positives among this set. Without this additional test a system could not distinguish between a worm and a piece of content that frequently occurs between two computers – for example a mail client sending the same user name repeatedly as it checks for new mail on the mail server regularly.

To quantify address dispersion one must count the *distinct* source IP addresses and destination IP addresses associated with each piece of content suspected of being generated by a worm (note that this is different from the previous content prevalence problem which only requires estimating the *repetitions* of each distinct string). While one could simply count source-destination address pairs, counting the source and destination addresses independently allows finer distinctions to be made. For example, mail messages sent to a popular mailing list are associated with many source-destination address pairs, but with only two sources – the mail server of the original sender of the message and the mail server running the list.

While it is possible to count IP addresses exactly using a simple list or hash table, more efficient solutions are needed if there are many pieces of content suspected of being generated by worms. Our solution is to trade off some precision in these counters for dramatic reductions in memory requirements. Our first approach was to appropriate the *direct bitmap* data structure originally

developed for approximate flow counting [46, 11]. Each content source is hashed to a bitmap, the corresponding bit is set, and an alarm is raised when the number of bits set exceeds a threshold. For example, if the dispersion threshold T is 30, the source address is hashed into a bitmap of 32 bits and an alarm is raised if the number of bits set crosses 20 (the value 20 is calculated analytically to account for hash collisions). This approach has minimal memory requirements, but in exchange it loses the ability to estimate the actual values of each counter – important for measuring the rate of infection or prioritizing alerts. While other techniques such as probabilistic counting [12] and multiresolution bitmaps [11] can provide accurate counts they require significantly more memory. For example a multiresolution bitmap requires 512 bits to count to 1 million.

Instead, we have invented a counting algorithm that leverages the fact that address dispersion continuously increases during an outbreak. Using this observation we devise a new, compact data structure, called a *scaled bitmap*, that accurately estimates address dispersion using five times less memory than existing algorithms.

The scaled bitmap achieves this reduction by subsampling the range of the hash space. For example, to count up to 64 sources using 32 bits, one might hash sources into a space from 0 to 63 yet only set bits for values that hash between 0 and 31 – thus ignoring half of the sources. At the end of a fixed measurement interval, this subsampling is adjusted by scaling the resulting count to estimate the true count (a factor of two in the previous example). Generalizing, we track a continuously increasing count by simply increasing this scaling factor whenever the bitmap is filled. For example the next configuration of the bitmap might map one quarter of the hash space to a 32 bit bitmap and scale the resulting count by four. This allows the storage of the bitmap to remain constant across an enormous range of counts.

However, once the bitmap is scaled to a new configuration, the addresses that were active throughout the previous configuration are lost and adjusting for this bias directly can lead to double counting. To minimize these errors, the final scaled bitmap algorithm, shown in Figure 3, uses multiple bitmaps ($numbmps = 3$ in this example) each mapped to progressively smaller and smaller portions of the hash space. To calculate the count, the estimated number of sources hashing to each bitmap are added, and then this sum is divided by the fraction of the hash space covered by all the bitmaps. When the bitmap covering the largest portion of the hash space has too many bits set to be accurate, it is advanced to the next configuration by recycling it: the bitmap is reset and then mapped to the next slice of the hash space (Figure 4). Consequently, each bitmap covers half the hash space covered by its predecessor. The first bitmap,

```

UpdateBitmap(IP)
1 code = Hash(IP)
2 level = CountLeadingZeroes(code)
3 bitcode = FirstBits(code << (level+1))
4 if (level ≥ base and level < base+numbmps)
5   SetBit(bitcode, bitmaps[level-base])
6   if (level == base and CountBitsSet(bitmaps[0]) == max)
7     NextConfiguration()
8   endif
9 endif

ComputeEstimate(bitmaps, base)
1 numIPs=0
2 for i=0 to numbmps-1
3   numIPs=numIPs+b ln(b/CountBitsNotSet(bitmaps[i]))
4 endfor
5 correction=
6   2(2base - 1)/(2numbmps - 1) · b ln(b/(b - max))
7 return numIPs · 2base/(1 - 2-numbmps) + correction

```

Figure 3: A scaled bitmap uses $numbmps$ bitmaps of size b bits each. The bitmaps cover progressively smaller portions of the hash space. When the bitmap covering the largest portion of the hash space gets too full to be accurate (the number of bits set reaches max), we advance to the next configuration by “recycling” the bitmap (see Figure 4). To compute an estimate of the number of distinct IP addresses, we multiply a estimate of the number of addresses that mapped to the bitmaps by the inverse of the fraction of the hash space covered by the bitmaps. A correction is added to the result to account for the IP addresses that were active in earlier configurations, while the current bitmaps were not in use at their present levels.

the one covering the largest portion of the hash space, is the most important in computing the estimate, but the other bitmaps provide “memory” for counts that are still small and serve to minimize the previously mentioned biases. Consequently, not much correction is needed when these bitmaps become the most important. Formally, we can prove that the maximum ratio between the bias of the algorithm and the number of active addresses is $2/(2^{numbmps} - 1)$ [36].

Overall, this new technique allows us to count sources and destinations quite accurately using only 3 bitmaps with roughly 5 times less memory than previously known techniques [12, 11]. This is critical for practical scaling because it reduces the system’s sensitivity to the effectiveness of the low-pass filter provided by the content prevalence test.

5.3 CPU scaling

Using multistage filters to detect content prevalence and scaled bitmaps to estimate address dispersion decreases memory usage and limits the amount of processing. However, each payload string still requires significant processing. In our prototype implementation (detailed in Section 6), the CPU can easily manage processing each packet payload as a single string, but when applying Rabin fingerprints, the processing of every substring of length β can overload the CPU during high traffic load. For example, a packet with 1,000 bytes of payload and $\beta = 40$, requires processing 960 Rabin fingerprints. While computing the Rabin fingerprints themselves incurs overhead, it is the three order of magnitude

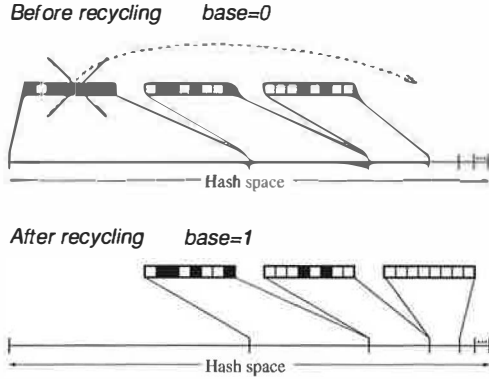


Figure 4: When the bitmap covering the largest portion of the hash space fills up, it is recycled. The bitmap is cleared and it is mapped to the largest uncovered portion of the hash space which is half the size of the portion covered by the bitmap right-most before recycling. Recycling increments the variable *base* (see Figure 3) by one.

increase in the number of content sifting operations that exceeds the capacity of our current CPU. While a faster CPU might solve this problem for a given traffic profile, the possibility of traffic surges and denial-of-service attacks on a sensor produce the same problem again. We believe that a security device should not fail in these circumstances but instead smoothly scale back functionality to match capacity – still performing the same functions but perhaps with reduced fidelity or responsiveness.

The obvious approach to address this problem is via dynamic sampling. However, randomly sampling which substrings to process could cause us to miss a large fraction of the occurrences of each substring and thus delay the generation of a worm's signature. Instead, we use *value sampling* [20] and select only those substrings for which the fingerprint matches a certain pattern (e.g. the last 6 bits of the fingerprint are 0). Consequently, the algorithm will systematically ignore some substrings, but track all occurrences of others. However, if a worm contains even a single tracked substring, it will be detected as promptly as without the sampling. For example, if f is the fraction of the tracked substrings (e.g. $f = 1/64$ if we track the substrings whose Rabin fingerprint ends on 6 0s), then the probability of detecting a worm with a signature of length x is $p_{track}(x) = 1 - e^{-f(x/\beta+1)}$.

Since Rabin fingerprint are randomly distributed themselves, the probability of tracking a worm substring of length β is f . Thus, the probability of missing the worm is $p_{miss}(\beta) = 1 - f$. The probability of not tracking the worm is the probability of not tracking any of its substrings. If the worm signature has length x , it has $x/\beta + 1$ substrings of length β . Assuming that no substring of length β repeats in the signature, the probability of not tracking the worm is $p_{miss}(x) = (1 - f)^{x/\beta+1} \approx e^{-f(x/\beta+1)}$. For example with $f = 1/64$ and $\beta = 40$, the probability of tracking a worm with a signature of

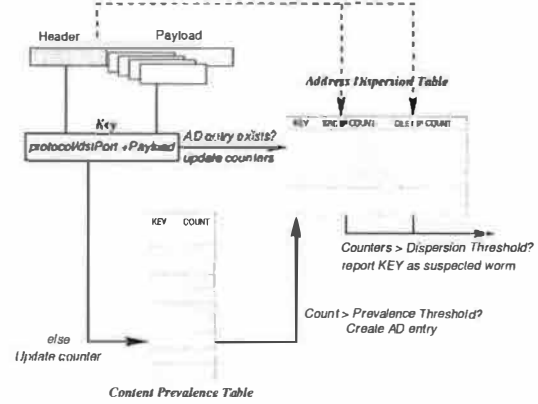


Figure 5: Content Sifting Algorithm as used in EarlyBird.

100 bytes is 55%, but for a worm with a signature of 200 bytes it increases to 92%, and for 400 bytes to 99.64%.

The sampling value f represents a tradeoff between processing and the probability of missing a worm; processing decreases linearly with f and the length of the invariant content required increases linearly with f . Note that all current worms have had invariant content of at least 400 bytes, for which the probability of false negatives is at most 0.36%. Our user-space software implementation requires $f = 1/64$ to keep up with roughly 200Mbps of traffic on a Gigabit Ethernet interface. Finally, since the parameters of the Rabin fingerprint algorithm p and M are not known, the worm writer cannot determine which strings will not be sampled in advance.

5.4 Putting it together

Figure 5 depicts the content sifting algorithm implemented in the EarlyBird prototype. As each packet arrives, its content (or substrings of its content) is hashed and appended with the protocol identifier and destination port to produce a content hash code. In our implementation, we use a 32-bit Cyclic Redundancy Check (CRC) as a packet hash and 40-byte Rabin fingerprints for substring hashes. Each Rabin fingerprint is subsampled with $f = 1/64$. The resulting hash codes are used to index the address dispersion table. If an entry already exists (the content has been determined to be prevalent) then the address dispersion table entries for source and destination IP addresses (implemented as scaled bitmaps) are updated. If the source and destination counts exceed the dispersion threshold, then the content string is reported.

If the content hash is not found in the dispersion table, it is indexed into the content prevalence table. In our implementation, we use four independent hash functions of the content hash to create 4 indexes into four counter arrays. Using the *conservative update* optimization, only the smallest among the four counters is incremented [10]. If all four counters are greater than the prevalence threshold, then a new entry is made in the ad-

dress dispersion table – with high probability, the content has appeared frequently enough to be a candidate worm signature. Pseudocode for the main loop of the EarlyBird system is shown in Figure 5.

```

ProcessPacket()
1 InitializeIncrementalHash(payload,payloadLength,dstPort)
2 while (currentHash=GetNextHash())
3   if (currentADEntry=ADEntryMap.Find(currentHash))
4     UpdateADEntry(currentADEntry,srcIP,dstIP,packetTime)
5     if ( (currentADEntry.srcCount > SrcDispTh)
        and (currentADEntry.dstCount > DstDispTh) )
6       ReportAnomalousADEntry(currentADEntry,packet)
7   endif
8 else
9   if ( MsfIncrement(currentHash) > PrevalenceTh)
10    newADEntry=InitializeADEntry(srcIP,dstIP,packetTime)
11    ADEntryMap.Insert(currentHash,newADEntry)
12  endif
13 endif
14 endwhile

```

Figure 6: The EarlyBird loop performed on every packet. When the prevalence threshold is exceeded, dispersion counting is done by creating an ADentry. ADentry contains the source and destination bitmaps and the scale factors required for the scaled bitmap implementation.

The content prevalence table sees the most activity in the system and serves as a high-pass filter for frequent content. The multi-stage filter data structure is cleared on a regular interval (60 seconds in our implementation). By contrast, the address prevalence table has typically fewer values – only those strings exceeding the prevalence threshold – and can be garbage collected over longer time scales (even hours).

Each of these mechanisms can be implemented at high speeds in either software or hardware, with relatively modest memory requirements as we quantify in the next section. Moreover, our approach makes no assumptions about the point of deployment, whether at the endpoint, edge, or core. However the optimal parameters settings may depend on the point of deployments. In Section 6 we empirically explore the parameter settings used by our EarlyBird prototype.

6 Experience

Based on the content sifting algorithm just described, we have built a prototype system which has been in use on the UCSD campus for over eight months. In this section, we describe our overall system design, the implementation and experimental environment, our initial experiments exploring the parameter space of the content sifting algorithm, our evaluation of false positives and false negatives, and our preliminary results in finding live worms at our site.

6.1 System design

The EarlyBird system consists of two major components: *Sensors* and an *Aggregator*. Each sensor sifts through traffic on configurable address space “zones” of responsibility and reports anomalous signatures. The aggregator

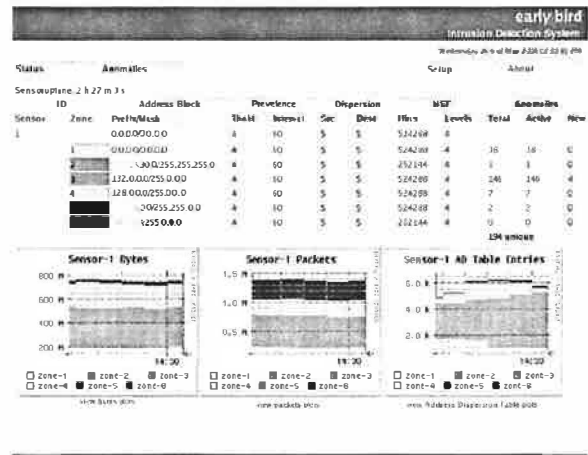


Figure 7: A screenshot of the main screen of the EarlyBird user interface. Each zone is labeled by a prefix and shows the current anomalies (worms), and prevalence/dispersion parameters which can be changed by the user. More detailed screens show detailed counts for each anomaly, as shown for Sasser in Figure 12.

coordinates real-time updates from the sensors, coalesces related signatures, activates any network-level or host-level blocking services and is responsible for administrative reporting and control. Our implementation is written in C and the aggregator also uses the MySQL database to log all events, the popular rrd-tools library for graphical reporting, and PHP scripting for administrative control. A screenshot of the main screen of the EarlyBird user interface showing zones and a summary of the current system activity is shown in Figure 7.

Finally, in order to automatically block outbreaks, the EarlyBird system automatically generates and deploys precise content-based signatures formatted for the Snort-inline intrusion prevention system [1]. A sample such signature for Kibvu.B is shown below.

```

drop tcp $HOME_NET any -> $EXTERNAL_NET 5000
(msg:"2712067784 Fri May 14 03:51:00 2004";
rev:1; content:"|90 90 90 90 4d 3f e3 77 90
90 90 90 ff 63 64 90 90 90 90 90|");

```

6.2 Implementation and environment

The current prototype Earlybird sensor executes on a 1.6Ghz AMD Opteron 242 1U server configured with a standard Linux 2.6 kernel. The server is equipped with two Broadcom Gigabit copper network interfaces for data capture. The EarlyBird sensor itself is a single-threaded application which executes at user-level and captures packets using the popular libpcap library. The system is roughly 5000 lines of code (not including external libraries) with the bulk of the code dedicated to self-monitoring for the purpose of this paper. The scalable implementation itself is a much smaller fraction of this code base. In its present untuned form, EarlyBird sifts through over 1TB of traffic per day and is able to keep up with over 200Mbps of continuous traffic when

using Rabin fingerprints with a value sampling probability of $1/64$ (and at even higher speeds using whole packet CRCs).

The experiments in the remainder of this paper are based on data collected from a Cisco Catalyst router configured to mirror all in-bound and out-bound traffic to our sensor (Earlybird currently makes no distinction between incoming and outgoing packets). The router manages traffic to and from roughly 5000 hosts, primarily clients, as well as all traffic to and from a few dedicated campus servers for DNS, SMTP/POP/IMAP, NFS, etc. The measured links experience a sustained traffic rate of roughly 100Mbps, with bursts of up to 500Mbps.

6.3 Parameter tuning

The key parameters used by our algorithm are the content prevalence threshold (currently 3), the address dispersion threshold (currently 30 sources and 30 destinations), and the time to garbage collect address dispersion table entries (currently several hours). We now describe the rationale behind these initial choices.

Content prevalence threshold: Figure 8 shows the distribution of signature repetitions on a trace for different hash functions. For example, using a 60 second measurement interval and a whole packet CRC, over 97 percent of all signatures repeat two or fewer times and 94.5 percent are only observed once. Using a finer grained-content hash or a longer measurement interval increases these numbers even further. However, to a first approximation, all reasonable values of these parameters reveal that very few signatures ever repeat more than 3 times. Recall that the principal benefit provided by the content prevalence table is to remove from consideration the enormous number of substrings which appear rarely and therefore are not possible worm signature candidates. We have repeated these experiments on several datasets at differing times and observed the same pattern. Consequently, for the remainder of this paper we use a prevalence threshold of 3.

Address dispersion threshold: Once a signature has passed the prevalence threshold it is still unlikely that it represents a worm. Figure 9 shows the number of distinct signatures found, as a function of time, for different address dispersion thresholds. For example, after 10 minutes there are over 1000 signatures (note the log scale) with a low dispersion threshold of 2 – meaning that the same string has been observed in packets with two different source IP addresses *and* two different destination IP addresses. However, as the dispersion threshold is increased, the number of such strings decreases dramatically. By the time a threshold of 30 is reached, there are only 5 or 6 prevalent strings meeting the dispersion criteria and the increase in this number is very slow over time. In this particular trace, two of these strings represent live

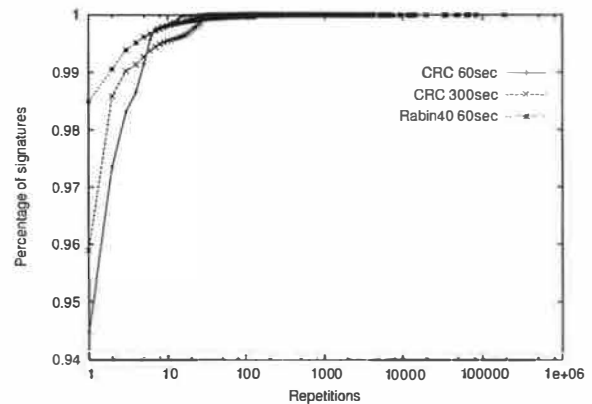


Figure 8: Cumulative distribution function of content signatures for different hash functions. This CDF is computed from the set of repetitions found in each measurement interval over a period of 10 minutes. Note that the y axis is artificially truncated to show more detail.

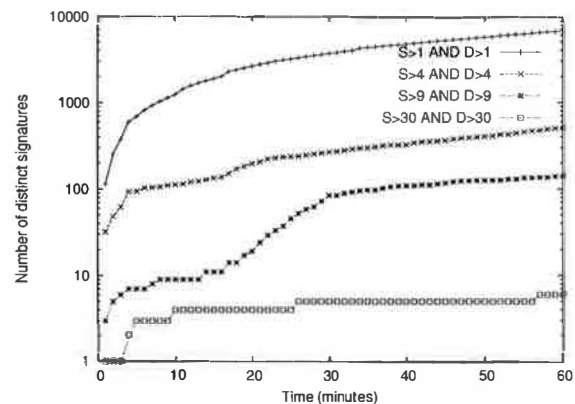


Figure 9: Number of distinct signatures detected over time for different address dispersion thresholds.

worms and the others are benign but consistently reoccurring strings that are post-filtered by a whitelist.

Note that there is an inherent tradeoff between the speed of detecting a new worm and the likelihood of a false positive. By using a lower dispersion threshold one can respond to a worm more quickly, but it is increasingly likely that many such signatures will be benign. For example, we find that the Slammer worm signature is detected within one second with an address dispersion threshold of 2, yet takes up to 5 seconds to discover using the more conservative threshold of 30. At the same time there are two orders of magnitude more signatures that will be reported with the lowest dispersion threshold – most of which will likely be false positives.

Garbage collection: The final key parameter of our algorithm is the elapsed time before an entry in the address dispersion table is garbage collected. The impact of this setting is shown in Figure 10. When the timeout is set to 100 seconds, then almost 60 percent of all signatures are garbage collected before a subsequent update

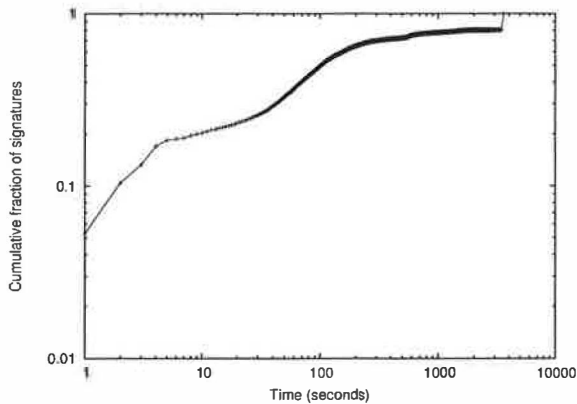


Figure 10: Log-scale cumulative distribution function of the maximum time period between updates for entries in the address dispersion table.

occurs – possibly preventing the signature from meeting the dispersion threshold and being reported. However, by a timeout of 1000 seconds, this number is reduced to roughly 20 percent of signatures. However, since the total number of signatures in the address dispersion table is always fairly small (roughly 25,000) we can comfortably maintain a timeout of several hours.

6.4 Performance

In Section 5 we described mechanisms such as multi-stage filters to reduce memory, and mechanisms such as value sampling to reduce CPU time. In this section, we briefly evaluate the performance of our prototype EarlyBird sensor in terms of processing time and memory.

Processing Time: To measure overhead, we instrumented the interface of each component to count elapsed CPU cycles. Because these counters are measured on a live system with varying packet sizes, and some functions (e.g., computing the Rabin hash) depend on packet size, we report the average and standard deviation over several million packets.

The top of Table 1 shows the overhead (in microseconds) incurred by each individual component of the EarlyBird algorithm as shown in Figure 6. The most significant operations are the initial Rabin fingerprint, accessing the multistage filter and creating a new Address Dispersion Table entry (dominated by the cost of malloc in this implementation). The Rabin algorithm is highly optimized based on Manber’s original code (as modified by Neil Spring), and incrementally amounts to a multiply and a mask (AND) operation. It seems difficult to optimize this particular hash function further in software, but it would be easy to implement in hardware. Similarly, a hardware implementation of multistage filters can use parallel memories to reduce the lookup time in proportion to the number of filter stages.

The bottom of Table 1 shows the overall processing time (in microseconds) taken by EarlyBird to process a

	Mean	Std. Dev.
Component wise breakdown		
Rabin Fingerprint		
First Fingerprint (40 bytes)	0.349	0.472
Increment (each byte)	0.037	0.004
Multi Stage Filter		
Test & Increment	0.146	0.049
AD Table Entry		
Lookup	0.021	0.032
Update	0.027	0.013
Create	0.252	0.306
Insert	0.113	0.075
Overall Packet		
Header Parsing & First Fingerprint	0.444	0.522
Per-byte processing	0.409	0.148
Overall Packet with Flow-Reassembly		
Header Parsing & Flow maintenance	0.671	0.923
Per-byte processing	0.451	0.186

Table 1: This table shows overhead (in microseconds) incurred by each of the individual operations performed on a packet. The mean and standard deviation are computed over a 10 minute interval (25 million Packets). This table represents raw overheads before sampling. Using 1 in 64 value sampling, the effective mean per byte processing time reduces to 0.042 microseconds.

packet. Without the use of value sampling, on an average it takes approximately 0.44 microseconds to parse the packet header and compute the first hash from the packet payload. Additionally for each byte in the packet payload EarlyBird adds an additional processing overhead of 0.409 microseconds on average. Utilizing (1 in 64) value sampling, as described in Section 5, brings down the average per-byte time to under 0.042 microseconds. This equates to 0.005 microseconds per bit, or a 200 Mbps line rate. We confirmed this by also examining the packet drop rate and comparing the output packet rate of the router, and the input packet rate seen by the system: at a sampling rate of 1 in 64 there are almost no dropped packets during 200Mbps load, but at smaller sampling rates there were significant numbers of dropped packets for equivalent input. In hardware, given the same value sampling rate, assuming that multiplies can be pipelined, and that the multistage filter memories and address dispersion tables operate in parallel, there is no reason why the algorithm should not scale with memory speeds even up to 40 Gbps.

Memory Consumption: The major memory hog is the content prevalence table, implemented using multi-stage filters with 4 stages, with each stage containing 524288 bins, and where each bin is 8 bits, for a total of 2MB. While this number of bins may appear to be large, recall that we are using a small prevalence threshold, and the amount of memory is still dramatically smaller than what would be required to index all content substrings.

The other major component of memory usage is the Address Dispersion Table which, in our experience, has

between 5K and 25K entries of 28 bytes each. All 25,000 of the Address Dispersion table entries combined utilize well under a Megabyte of memory. The other components use a negligible amount of memory. Thus the core EarlyBird function currently consumes less than 4 Mbytes of memory. If the content prevalence threshold was made higher (as could be done in deployments nearer the core of the network), the memory needs of the multistage filters and address dispersion tables will go down dramatically, allowing potential on-chip implementations; current FPGAs allow 1 Mbyte of on-chip SRAM, and custom chips allow up to 32 Mbytes.

6.5 Trace-based verification

In this subsection, we report on experimental results for false positives and false negatives. While a key feature of our system is that it runs on *live* network traffic, for these experiments we replayed a captured trace in real-time to support comparisons across runs. We report on some of our live experience in the next subsection.

False Positives: Any worm detection device must contend with false positives. Figure 11 shows the prevalence of different signatures over time that meet the dispersion threshold of 10 (we set the threshold in this experiment lower than the current parameter setting of 30 used in our live system to produce more signatures), while the signatures themselves are listed in Table 2. The two most active signatures belong to the Slammer and Opaserv worms, followed by a pervasive string on TCP port 445 used for distributed port scanning, and the Blaster worm. The remaining signatures fall into two categories: those that are likely worms or distributed scans (likely, due to the high degree of such traffic sent to these ports from outside the LAN) and a few strings that represent true false positives and arise from particular protocol structures.

Over longer live runs we have found two principal sources of false positives: common protocol headers and unsolicited bulk email (SPAM). In the former category, over 99 percent of all false positives result from distinct SMTP header strings or HTTP user-agent or content-type strings. We have observed over 2000 of these combinations in practice, all of which are easily whitelisted procedurally via a protocol parser. It is critical to do so however, since automatically blocking or rate-limiting even a significant subset of HTTP traffic could have disastrous consequences. The other principal source of false positives are SPAM e-mails which can exceed address dispersion thresholds due to the use of distributed mailers and mail relays. While these are far more difficult to whitelist since many e-mail viruses also propagate via TCP port 25, the effect of their interdiction is far more benign as well. Moreover, false positives arising from SPAM are bursty in practice since they coincide with a mass mail-

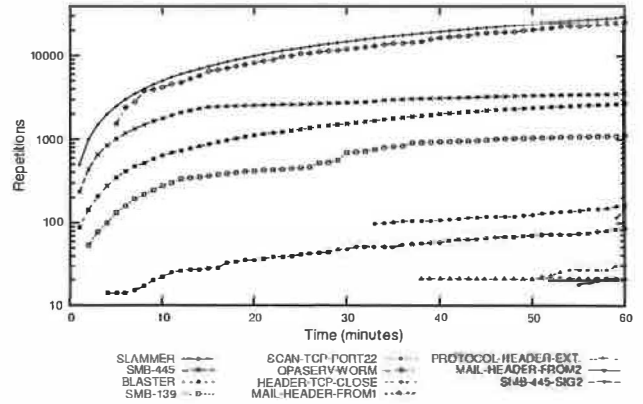


Figure 11: Count of worm-like signatures as a function of time. This graph uses a content prevalence threshold of 3 and an address dispersion threshold of 10 (i.e. $d > 10$ and $s > 10$).

ing cycle. Consequently, even without additional policy, countermeasures for such traffic tends to automatically self-limit.

We have also seen a few false positives that simply reflected unique aspects of a few popular protocol implementations. For example, the most common signatures in our data (not shown above) are strings of all zeros and all ones in base64 encoding (caused by a popular mail reader's text encoding) and we exclude these from consideration statically. Similarly, the HEADER-TCP-CLOSE signature shown above, is a string "tcp-close, during connect" that is included in TCP RST packets sent by the MAC OS X operating system. The system itself is sufficiently common that the address dispersion and content prevalence criteria are met. However, there are sufficiently few of these, even over extended periods of analysis, that a handful of static whitelist entries have been sufficient to remove them.

Finally, over long traces we have observed one source of false positives that defies easy analysis. In particular, popular files distributed by the BitTorrent peer-to-peer system can satisfy the content prevalence and address dispersion criteria during their peak periods of popularity. This does not seem to happen in practice with other peer-to-peer systems that rely on whole-file download, but BitTorrent's file striping creates a many-to-many download profile that mimics that of a worm.

In general, all of our false positives appear consistently across multiple trials over different time periods. This leads us to believe that most are relatively stable and reflect a small number of pathologies. Consequently, in live use we excluded such signatures in a small "whitelist" which is used to post-filter signature reports.

False negatives: Since our experiments have been run in an uncontrolled environment it is not possible to quantitatively demonstrate the absence of false negatives. However, a strong qualitative indication is that Earlybird running live detected every worm outbreak reported on

Label	Service	Sources	Dests
SLAMMER	UDP/1434	3328	23607
SCAN-TCP-PORT22	TCP/22	70	53
MAIL-HEADER-FROM	TCP/25	12	11
SMB-139	TCP/139	603	378
SMB-445	TCP/445	2039	223
HEADER-TCP-CLOSE	TCP/80	33	136
MAIL-HEADER-FROM2	TCP/25	13	14
PROTOCOL-HEADER-EXT	TCP/80	15	24
BLASTER	TCP/135	1690	17
OPASERV-WORM	UDP/137	180	21033
SMB-445-SIG2	TCP/445	11	145

Table 2: Summary signatures reported using an address dispersion threshold of 10.

public security mailing lists (including BugTraq, Full-Disclosure, and snort-signatures) during our period of operation. We also checked for false negatives by comparing the trace we used against a Snort rulebase including over 340 worm and worm-related signatures aggregated from the official Snort distribution as well as the snort-signatures mailing list. We found no false negatives via this method, although the Snort system alerted on a number of instances that were not worms.

6.6 Inter-packet signatures

As described, the content-sifting algorithm used in EarlyBird does not keep any per-flow state and can therefore only generate content signatures that are fully contained within a single packet. Thus an attacker might evade detection by splitting an invariant string into pieces one byte smaller than β – one per packet.

We have extended the content sifting algorithm to detect such simple evasions at the cost of per flow state management. While there are many approaches to flow reassembly, our initial design point is one that trades the fully general reassembly for reduced overhead by exploiting Earlybird's use of incremental fingerprints. Thus, for each flow we maintain a circular buffer containing the last 40 bytes, as well as the Rabin fingerprint for this string and some book-keeping state for managing flow expiration. Using this data we are able to continue the signature matching process from each packet to its successor (in fact the computation cost is reduced as a result, since the expensive per-packet Rabin fingerprint is only necessary for the first packet in a flow). Currently, we manage flows first-come first-served via a flow cache, although it is easy to bias this allocation to favor sources exhibiting abnormal activity such as port-scanning or accessing unused address space [34, 35, 37, 24]. It should be clear that a sophisticated worm or virus which subverts the host operating system will be able to reorder or arbitrarily delay packets in a way that evades this approach. We describe the challenges of more complex

evasions in Section 7. We briefly evaluated the performance impact of this extension and found that using a flow cache of 131072 elements (7MB in total) the average cost for processing the first packet of a flow is increased by 0.227 microseconds and the average per-byte cost is increased by 0.042 (absolute numbers and associated standard deviations are reported in Table 1).

6.7 Live experience with EarlyBird

In addition to the worms described above, Earlybird has also detected precise signatures for variants of CodeRed, the MyDoom mail worm and most recently for the Sasser, and Kibvu.B worm. In the case of new worms such as Kibvu.B and MyDoom, Earlybird reported signatures long before there were public reports of the worm's spread – let alone signatures available – and we were able to use these signatures to assist our network operations staff in tracking down infected hosts.

While we have experience with all recent worms, we limit ourselves to describing our experience with two recent outbreaks, Sasser and Kibvu.B.

Sasser: We detected Sasser on the morning of Saturday May 1st, 2004. Though we cannot claim to be the first ones to detect Sasser, we certainly did detect it before signatures were made available by the various anti-virus vendors. Part of the reason for us not detecting Sasser earlier is because all inbound traffic destined to port 445 is dropped at the upstream router and thus we could only use strictly internal traffic to make an identification. Figure 12 shows a screenshot of the live EarlyBird system tracking the rate in growth of infected Sasser hosts and their attempts to infect others in the UCSD network.

Kibvu.B: Kibvu.B is a recent worm that Earlybird detected on Friday May 14th, 2003 at 3:08AM PDT. In contrast to other outbreaks, Kibvu.B was extremely subdued (perhaps because it targeted a two year old vulnerability that was less prevalent in the host population). In the 40 minute window following our first recorded instance of the worm, there were a total of 37 infection attempts to 30 unique destinations, allowing us to trigger based on our 30-30 dispersion threshold. The Kibvu.B experience suggests that simply utilizing content prevalence as a metric as in [16] may not be sufficient; address dispersion is essential. We have provided a signature for this worm in section 6.1.

7 Limitations and Extensions

While we have been highly successful with our prototype system, we recognize a number of limitations, potential challenges and problems facing those building a complete worm defense system. In this section we discuss these and discuss current extensions that we are adding to our system to address these issues.

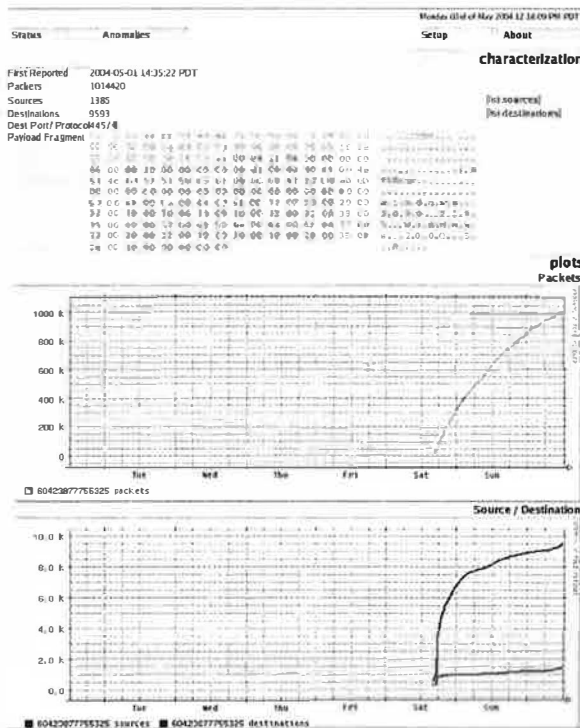


Figure 12: A detailed screen showing EarlyBird's capture of the Sasser outbreak. The anomaly is labeled 6042387755325 because at the time of discovery the anomaly was not named. The top of the screen-shot shows the signature and the number of sources and destinations. The middle of the screen shot shows a time-series plot of the packets containing the signature over roughly 2 days. The bottom of the screen shot shows a time-series plot of unique destinations (top curve) and unique sources (bottom curve) with the content. The destinations are much larger because the same sources are attempting to infect a large number of destinations.

7.1 Variant content

If content sifting were to be widely deployed this could create an incentive for worm writers to design worms with little or no invariant content. For example, polymorphic viruses encrypt their content in each generation and so-called “metamorphic viruses” have even demonstrated the ability to mutate their entire instruction sequence with semantically equivalent, but textually distinct, code. This is an extremely challenging problem that is currently addressed by antivirus vendors using controlled emulation [2] and procedural signatures. While many of these subterfuges are trivially detectable (e.g. since polymorphic decryption code may be itself invariant), and others can be detected by modifying our content sifting approach to identify textually “similar” content – in the limit this threat is a fundamental one. As part of future work we are investigating hybrid pattern matching approaches that quickly separate non-code strings (identifiable by unavoidable terminating instruction sequences) from potential exploits – and focus complex analysis only on those sequences which pose a threat.

Other problems are presented by compression. While

existing self-encoding viruses maintain an invariant decoding routine, a worm author might choose to reuse a common code sequence (e.g., such as one used for a popular self-decompressing executable format, like ZIP executables). Using this string as a worm signature could produce many false positives for content using the same sequence. Finally, several vulnerabilities have been found in popular implementations of encrypted session protocols such as ssh and SSL. Worms exploiting such vulnerabilities can *opportunistically* make use of the per-session encryption offered by these services. As a result, content-oriented traffic analysis, like that described in this paper, would be impossible. The same problem is posed by widespread deployment of end-to-end IPSEC and virtual private networks (VPNs). This problem appears to be fundamental. Indeed if such deployments become widespread much of the current security market (especially including current intrusion detection systems) will have to be rethought.

7.2 Network evasion

Just as attackers may attempt to evade content sifting algorithms by creating metamorphic worms, they may also attempt to evade our monitor through traditional IDS evasion techniques [30]. While we discussed the problem of *flow reassembly* earlier, a sophisticated attacker might send overlapping IP fragments or TCP segments to create a network-level polymorphism. To address this issue requires *traffic normalization*, in which datagrams are re-assembled in a single consistent fashion [14]. However, in its full generality, this approach requires far more per-flow state and per-packet computation than mere flow reassembly and therefore may not scale well without further performance-enhancing techniques. An alternative we are considering is to simply filter such obviously odd-ball packets – at the cost of some impact on sites which actually depend on non-standard TCP segmentation or IP fragmentation implementations.

Finally, incidental network evasion may occur if the assumptions underlying the address dispersion threshold are violated. For example, if a worm requires only a single packet for transmission then the attacker could spoof the source address so all packets appear to originate from the same source. While such evasions are easy to detect, it requires special purpose code outside the general content sifting framework.

7.3 Extensions

In addition to the potential challenges posed by malicious actors, there are a number of additional improvements that could be made to our system even in the current environment. For example, while we have experienced that given parameter settings appear to provide consistent results on our link across time, our settings were

themselves based on measurement and experimentation. We believe they are sensitive to the number of live hosts interdicted by the monitor, but exactly how remains an open question. In the next generation of our system, we plan to use techniques similar to [10] to “autotune” EarlyBird’s content sifting parameters for a given environment.

Finally, while most worms to date have sought to maximize their growth over time, it is important to address the issue of slow worms as well. In our current prototype, worms which are seen less frequently than every 60 seconds have no hope of registering. One method to address this limitation within our system is to maintain triggering data across multiple time scales. Alternatively, one might deploy a hybrid system, using Earlybird to intercept high-speed outbreaks worms and host-based intrusion detection or large-scale honeypots to detect slowly spreading pathogens. Indeed, even small detection probabilities can eliminate the stealthy advantage of slow worms and thus the incentive for deploying them.

7.4 Containment

Our current system reports the suspected worm signatures, but can be configured to generate Snort signatures in a few seconds which can then be blocked by an online Snort deployment. We have been doing so on a small scale on a laboratory switch, and the system has blocked worm traffic based on the signatures we feed the blocker. Unfortunately, the policy for applying such a containment strategy can be quite complex. For example, since there is an inherent tradeoff between detection speed and false positives, as we discussed earlier, one reasonable policy is to temporarily rate-limit traffic matching signatures with only moderate address dispersion. If the signature is a false positive then it likely will never reach a higher level of dispersion and the rate-limit can be repealed. If it is a worm, then this conservative reaction will slow its spread and once its dispersion increases to a higher level the system can decide to drop all packets carrying the signature. However, this is just one such policy option and the question deserves additional attention.

Moreover, automated containment also provokes the issue of attackers purposely trying to trigger a worm defense – thereby causing denial-of-service on legitimate traffic also carrying the string. Thus, a clear area of research for us is to develop efficient mechanisms for comparing signatures with existing traffic corpus’ – to understand the *impact* of filtering such traffic before we do so. However, even this approach may fall short against a sophisticated attacker with prior knowledge of an unreleased document. In this scenario an attacker might coerce Earlybird into blocking the documents release by simulating a worm containing substrings unique only to the unreleased document.

7.5 Coordination

One of the key benefits of signature extraction is that a given signature can be shared. This provides a “network effect” because the more deployments are made of a system such as ours, the more value there is to all deployments because of sharing. This sharing in turn can reduce response times, since the first site to discover a new worm signature can share it immediately. A more aggressive possibility is to add this detection capability to core routers which can then spread the signatures to edge networks. The issue of coordination brings up substantial questions related to trust, validation and policy that will require additional research attention to address.

8 Conclusions

New worm outbreaks routinely compromise hundreds of thousands of hosts and despite the enormous recovery costs incurred for past worms, we have been extremely fortunate in the degree of restraint demonstrated by worm authors. Thus the need for an adequate defense against future worm episodes is self-evident.

In this paper, we have described an approach for real-time detection of unknown worms and automated extraction of unique content signatures. Our content sifting algorithm efficiently analyses network traffic for prevalent and widely dispersed content strings – behavioral cues of worm activity. We have demonstrated that content sifting can be implemented with moderate memory and computational requirements and our untuned software-based prototype has been able to process over 200Mbps of live traffic. While the security field is inherently an “arms race”, we believe that systems based on content sifting significantly raise the bar for worm authors. To wit, in our experience Earlybird has been able to detect and extract signatures for *all* contemporary worms and has also demonstrated that it can extract signatures for new, previously unknown, worms.

While we believe that EarlyBird can be a useful system in itself, we believe that the underlying method (maintaining state keyed by content signatures) may generalize to address a number of other interesting research problems. For example, we have found that slight modifications to Earlybird are able to detect large amounts of unsolicited bulk e-mail (SPAM) based on the same general principles as worm detection. Similarly, mass-intrusion attempts can also be revealed by this approach, as can denial-of-service attacks and peer-to-peer system activity.

Finally, the EarlyBird system demonstrates the feasibility of sophisticated wire-speed network security. While many industrial systems have only recently announced signature *detection* at Gigabit speeds, our experience with Earlybird suggests that signature *learning* at Gigabit speeds is equally viable. This leads us to hope

that other components of network security may also permit wire-speed implementation and allow security functions to be integrated – as a standard part of routers and switches – into the very heart of the network.

9 Acknowledgements

We would like to thank David Moore, Ramana Kompella and Geoff Voelker for their insightful discussions and Colleen Shannon, Jim Madden, Pat Wilson and David Visick for helping us understand the UCSD network. Finally, we would like to thank both the anonymous reviewers and our shepherd, David Wagner, for their constructive comments and suggestions. Support for this work was provided by NIST Grant 60NANB1D0118 and NSF Grant 0137102.

References

- [1] Snort: Open source network intrusion detection system. www.snort.org, 2002.
- [2] Carey Nachenberg. Method to analyze a program for presence of computer viruses by examining the opcode for faults before emulating instruction in emulator. U.S. Patent 5,964,889, Oct. 1999.
- [3] Cisco Systems, Inc. Cisco Security Agent ROI: Deploying Intrusion Protection Agents on the Endpoint. Cisco Technical Whitepaper.
- [4] F. Cohen. Computer Viruses — Theory and Experiments. In *Proceedings of the 7th DoD/NBS Computer Security Conference*, Sept. 1984.
- [5] F. Cohen. Computer Viruses — Theory and Experiments. *Computers and Security*, 6:22–35, 1987.
- [6] Colleen Shannon and David Moore. The Spread of the Witty Worm. *IEEE Security and Privacy*, 2(4), July 2004.
- [7] N. Duffield and M. Grossglauser. Trajectory sampling for direct traffic observation. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2000.
- [8] M. Erbschloe. Computer Economics VP Research Statement to Reuters News Service, Nov. 2001.
- [9] C. Estan, S. Savage, and G. Varghese. Automatically Inferring Patterns of Resource Consumption in Network Traffic. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2003.
- [10] C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2002.
- [11] C. Estan, G. Varghese, and M. Fisk. Bitmap algorithms for counting active flows on high speed links. In *Proceedings of the ACM Internet Measurement Conference*, Oct. 2003.
- [12] P. Flajolet and G. N. Martin. Probabilistic Counting Algorithms for Data Base Applications. *Journal of Computer and System Sciences*, 31(2):182–209, Oct. 1985.
- [13] P. B. Gibbons and Y. Matias. New Sampling-based Summary Statistics for Improving Approximate Query Answers. In *Proceedings of the ACM SIGMOD Conference*, June 1998.
- [14] M. Handley, V. Paxson, and C. Kreibich. Network Intrusion Detection: Evasion, Traffic Normalization and End-to-End Protocol Semantics. In *Proceedings of the USENIX Security Symposium*, Aug. 2001.
- [15] J. O. Kephart and W. C. Arnold. Automatic extraction of computer virus signatures. In *Proceedings of the 4th International Virus Bulletin Conference*, Sept. 1994.
- [16] K.-A. Kim and B. Karp. Autograph: Toward Automated Distributed Worm Signature Detection. In *Proceedings of the USENIX Security Symposium*, Aug. 2004.
- [17] C. Kreibich and J. Crowcroft. Honeycomb – Creating Intrusion Detection Signatures Using Honeypots. In *Proceedings of the USENIX/ACM Workshop on Hot Topics in Networking*, Nov. 2003.
- [18] J. Levin, R. LaBella, H. Owen, D. Contis, and B. Culver. The Use of Honeynets to Detect Exploited Systems Across Large Enterprise Networks. In *Proceedings of the 2003 IEEE Workshop on Information Assurance*, June 2003.
- [19] J. W. Lockwood, J. Moscola, M. Kulig, D. Reddick, and T. Brooks. Internet worm and virus protection in dynamically reconfigurable hardware. In *Proceedings of the Military and Aerospace Programmable Logic Device Conference*, Sept. 2003.
- [20] U. Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter Technical Conference*, 17–21 1994.
- [21] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. The Spread of the Sapphire/Slammer Worm. *IEEE Security and Privacy*, 1(4), July 2003.
- [22] D. Moore, C. Shannon, and J. Brown. Code-Red: A Case Study on the Spread and Victims of an Internet Worm. In *Proceedings of the ACM Internet Measurement Workshop*, Nov. 2002.
- [23] D. Moore, C. Shannon, G. Voelker, and S. Savage. Internet Quarantine: Requirements for Containing Self-Propagating Code. In *IEEE Proceedings of the INFOCOM*, Apr. 2003.
- [24] D. Moore, C. Shannon, G. Voelker, and S. Savage. Network Telescopes. Technical Report CS2004-0795, CSE Department, UCSD, July 2004.
- [25] D. Moore, G. M. Voelker, and S. Savage. Inferring Internet Denial-of-Service Activity. In *Proceedings of the USENIX Security Symposium*, Aug. 2001.
- [26] C. Morrow. BlackHole Route Server and Tracking Traffic on an IP Network. <http://www.secsup.org/Tracking/>.
- [27] A. Muthitacharoen, B. Chen, and D. Mazieres. A Low-bandwidth Network File System. In *Proceedings of the ACM SOSOP Conference*, Oct. 2001.
- [28] Network Associates Inc. McAfee Entercept Standard Edition. Product Datasheet.
- [29] V. Paxson. Bro: a System for Detecting Network Intruders in Real-time. In *Proceedings of the USENIX Security Symposium*, Jan. 1998.
- [30] T. H. Ptacek and T. N. Newsham. Insertion, Evasion and Denial-of-Service: Eluding Network Intrusion Detection. Technical report, Secure Networks Inc., Jan. 1998.
- [31] J. C. Rabek, R. I. Khazan, S. M. Lewandowski, and R. K. Cunningham. Detection of Injected, Dynamically Generated, and Obfuscated Malicious Code. In *Proceedings of the 2003 ACM Workshop on Rapid Malcode*, Oct. 2003.
- [32] M. O. Rabin. Fingerprinting by Random Polynomials. Technical Report 15-81, Center for Research in Computing Technology, Harvard University, 1981.
- [33] J. Rochlis and M. Eichen. With Microscope and Tweezers: The Worm from MIT's Perspective. *Communications of the ACM*, 32(6):689–698, June 1989.
- [34] S. Singh, C. Estan, G. Varghese, and S. Savage. Real-time Detection of Known and Unknown Worms. Technical Report CS2003-0745, CSE Department, UCSD, May 2003.
- [35] S. Singh, C. Estan, G. Varghese, and S. Savage. The EarlyBird System for Real-time Detection of Unknown Worms. Technical Report CS2003-0761, CSE Department, UCSD, Aug. 2003.
- [36] S. Singh, C. Estan, G. Varghese, and S. Savage. Accuracy bounds for the scaled bitmap data structure. Technical Report CS2004-0814, CSE Department, UCSD, Dec. 2004.
- [37] S. Singh, G. Varghese, C. Estan, and S. Savage. Detecting Public Network Attacks using Signatures and Fast Content Analysis. United States Patent Application.
- [38] A. C. Snoeren, C. Partridge, C. E. Jones, F. Tchakountio, S. T. Kent, and W. T. Strayer. Hash-based IP Traceback. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2001.
- [39] E. Spafford. The Internet Worm: Crisis and Aftermath. *Communications of the ACM*, 32(6):678–687, June 1989.
- [40] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2000.
- [41] S. Staniford. Containment of Scanning Worms in Enterprise Networks. to appear in the *Journal of Computer Security*, 2004.
- [42] S. Staniford, V. Paxson, and N. Weaver. How to Own the Internet in Your Spare Time. In *Proceedings of the USENIX Security Symposium*, Aug. 2002.
- [43] J. Twycross and M. M. Williamson. Implementing and Testing a Virus Throttle. In *Proceedings of the USENIX Security Symposium*, Aug. 2003.
- [44] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In *Proceedings of the ACM SIGCOMM Conference*, Aug. 2004.
- [45] N. Weaver. Personal communication, 2002.
- [46] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor. A Linear-Time Probabilistic Counting Algorithm for Database Applications. *ACM Transactions on Database Systems*, 15(2):208–229, 1990.

Understanding and Dealing with Operator Mistakes in Internet Services *

Kiran Nagaraja, Fábio Oliveira, Ricardo Bianchini, Richard P. Martin, Thu D. Nguyen

Department of Computer Science

Rutgers University, Piscataway, NJ 08854

{knagaraj, fabiool, ricardob, rmartin, tdnguyen}@cs.rutgers.edu

Abstract

Operator mistakes are a significant source of unavailability in modern Internet services. In this paper, we first characterize these mistakes by performing an extensive set of experiments using human operators and a realistic three-tier auction service. The mistakes we observed range from software misconfiguration, to fault misdiagnosis, to incorrect software restarts. We next propose to validate operator actions before they are made visible to the rest of the system. We demonstrate how to accomplish this task via the creation of a validation environment that is an extension of the online system, where components can be validated using real workloads before they are migrated into the running service. We show that our prototype validation system can detect 66% of the operator mistakes that we have observed.

1 Introduction

Online services, such as search engines, e-mail, work-group calendars, and music juke-boxes are rapidly becoming the supporting infrastructure for numerous users' work and leisure. Increasingly, these services are comprised of complex conglomerates of distributed hardware and software components, including front-end devices, numerous kinds of application packages, authenticators, loggers, databases, and storage servers.

Ensuring high availability for these services is a challenging task. First, frequent hardware and software upgrades keep these systems constantly evolving. Second, this evolution and the complexity of the services imply a large number of unforeseen interactions. Third, component failure is a common occurrence, since these services are typically based on commodity components for fast deployment and low cost. Given these factors, it

is not surprising that service failures occur frequently [12, 19, 20].

In this paper, we characterize and alleviate one significant source of service failures, namely *operator mistakes*, in the context of cluster-based Internet services. Several studies have shown that the percentage of service failures attributable to operator mistakes has been increasing over the last decade [12, 17, 20]. A recent study of three commercial services showed that operator mistakes were responsible for 19-36% of the failures, and, for two of the services, were the dominant source of failures and the largest contributor to time to repair [19]. An older study of Tandem systems also found that operator mistakes were a dominant reason for outages [11].

Our work begins with a set of live operator experiments that explore the nature of operator mistakes and their impact on the availability of a three-tier auction service [21]. In each experiment, an operator must either perform a scheduled maintenance task or a diagnose-and-repair task. The first category encompasses tasks such as upgrading software, upgrading hardware, and adding or removing system components. The second category encompasses experiments during which we inject a fault into the service and ask the operator to discover and fix the problem.

The operator experiments do *not* seek to cover all possible operator tasks or to achieve a complete statistical characterization of operator behavior. Rather, our goal is to characterize some of the mistakes that can occur during common operator tasks, and to gather detailed traces of operator actions that can be used to evaluate the effectiveness of techniques designed to either prevent or to mitigate the impact of operator mistakes. We are continuing our experiments to cover a wider range of operator tasks and to collect a larger sampling of behaviors.

So far, we have performed 43 experiments with 21 volunteer operators with a wide variety of skill levels. Our results show a total of 42 mistakes, ranging from software configuration, to fault misdiagnosis, to soft-

*This research was partially supported by NSF grants #EIA-0103722, #EIA-9986046, and #CCR-0100798.

ware restart mistakes. Configuration mistakes of different types were the most common with 24 occurrences, but incorrect software restarts were also common with 14 occurrences. A large number of mistakes (19) led to a degradation in service throughput.

Given the large number of mistakes the operators made, we next propose that services should *validate* operator actions before exposing their effects to clients. The key idea is to check the correctness of operator actions in a *validation environment* that is an extension of the online system. In particular, the components under validation, called *masked* components, should be subjected to realistic (or even live) workloads. Critically, their configurations should not have to be modified when transitioning from validation to live operation.

To demonstrate our approach and evaluate its efficacy, we have implemented a prototype validation framework and modified two applications, a cooperative Web server and our three-tier auction service, to work within the framework. Our prototype currently includes two validation techniques, trace-based and replica-based validation. Trace-based validation involves periodically collecting traces of live requests and replaying the trace for validation. Replica-based validation involves designating each masked component as a “mirror” of a live component. All requests sent to the live components are then duplicated and also sent to the mirrored, masked component. Results from the masked components are compared against those produced by the live component.

We evaluate the effectiveness of our validation approach by running a wide range of experiments with our prototype: (1) microbenchmarks that isolate its performance overhead; (2) experiments with human operators; (3) experiments with mistake traces; and (4) mistake-injection experiments. From the microbenchmarks, we find that the overhead of validation is acceptable in most cases. From the other experiments, we find that our prototype is easy to use in practice, and that the combination of trace and replica-based validation is effective in catching a majority of the mistakes we have seen. In particular, using detailed traces of operator mistakes, we show that our prototype would have detected 28 out of the 42 mistakes observed in our operator experiments.

In summary, we make two main contributions:

- We present detailed data on operator behavior during a large set of live experiments with a realistic service. Traces of all our experiments are available from <http://vivo.cs.rutgers.edu/>. This contribution is especially important given that actual data on operator mistakes in Internet services is not publicly available, due to commercial and privacy considerations. We also analyze and categorize the reasons behind the mistakes in detail.

- We design and implement a prototype validation framework that includes a realistic validation environment for dealing with operator mistakes. We demonstrate the benefits of the prototype through an extensive set of experiments, including experiments with actual operators.

We conclude that operators make mistakes even in fairly simple tasks (and with plenty of detailed information about the service and the task itself). We conjecture that these mistakes are mostly a result of the complex nature of modern Internet services. In fact, a large fraction of the mistakes cause problems in the interaction between the service components in the actual processing of client requests, suggesting that the realism derived from hosting the validation environment in the online system itself is critical. Given our experience with the prototype, we also conclude that validation should be useful for real commercial services, as it is indeed capable of detecting several types of operator mistakes.

The remainder of the paper is organized as follows. The next section describes the related work. Section 3 describes our operator experiments and their results. Section 4 describes the details of our validation approach and prototype implementation, and compares our approach with offline testing and undo in the context of the mistakes we observed. Section 5 presents the results of our validation experiments. Finally, Section 6 concludes the paper.

2 Related Work

Only a few papers have addressed operator mistakes in Internet services. The work of Oppenheimer *et al.* [19] considered the universe of failures observed by three commercial services. With respect to operators, they broadly categorized their mistakes, described a few example mistakes, and suggested some avenues for dealing with them. Here, we extend their work by describing all of the mistakes we observed in detail and by designing and implementing a prototype infrastructure that can detect a majority of the mistakes.

Brown and Patterson [6] have proposed “undo” as a way to rollback state changes when recovering from operator mistakes. Brown [5] has also performed experiments in which he exposed human operators to an implementation of undo for an email service hosted by a single node. We extend his results by considering a more complex service hosted by a cluster. Furthermore, our validation approach is orthogonal to undo in that we hide operator actions from the live service until they have been validated in a realistic validation environment. We discuss undo further in Section 4.6.

A more closely related technique is “offline testing” [3]. Our validation approach takes offline testing a step further by operating on components in a validation environment that is an extension of the live service. This allows us to catch a larger number of mistakes, as we discuss in Section 4.6.

The Microvisor work by Lowell *et al.* [16] isolates on-line and maintenance parts of a single node, while keeping the two environments identical. However, their work cannot be used to validate the interaction among components hosted on multiple nodes.

Our trace-based validation is similar in flavor to various fault diagnosis approaches [2, 10] that maintain statistical models of “normal” component behavior and dynamically inspect the service execution for deviations from this behavior. These approaches typically focus on the data flow behavior across the systems components, whereas our trace-based validation inspects the actual responses coming from components and can do so at various semantic levels. We also use replica-based validation, which compares the responses directly to those of “correct” live components.

Replication-based validation has been used before to tolerate Byzantine failures and malicious attacks, e.g. [8, 9, 13]. In this context, replicas are a permanent part of the distributed system and validation is constantly performed through voting. In contrast, our approach focuses solely on dealing with operator mistakes, does not require replicas and validation during normal service execution, and thus can be simpler and less intrusive.

Other orthogonal approaches to dealing with operator mistakes or reducing operator intervention have been studied [1, 4, 14]. For example, Ajmani *et al.* [1] eliminate operator intervention in software upgrades, whereas Kiciman *et al.* [14] automate the construction of correctness constraints for checking software configurations.

3 Operator Actions and Mistakes

In this section, we analyze the maintenance and diagnose-and-repair experiments that we performed with human operators and our three-tier auction service. We start by describing the experimental setup, the actual experiments, and the operators who volunteered to participate in our study. After that, we detail each experiment, highlighting the mistakes made by the operators.

3.1 Experimental setup

Our experimental testbed consists of an online auction service modeled after EBay. The service is organized into three tiers of servers: Web, application, and database tiers. We use two machines in the first tier running the

Apache Web server (version 1.3.27), five machines running the Tomcat servlet server (version 4.1.18) in the second tier and, in the third tier, one machine running the MySQL relational database (version 4.1.2). The Web server and application server machines are equipped with a 1.2 GHz Intel Celeron processor and 512 MB of RAM, whereas the database machine relies on a 1.9 GHz Pentium IV with 1 GB of RAM. All machines run Linux with kernel 2.4.18-14.

The service requests are received by the Web servers and may flow towards the second and third tiers. The replies flow through the same path in the reverse direction. Each Web server keeps track of the requests it sends to the application servers. Each application server maintains the soft state associated with the client sessions that it is currently serving. This state consists of the auctions of interest to the clients. All dynamic requests belonging to a session need to be processed by the same application server, thereby restricting load balancing. A heartbeat-based membership protocol is used to reconfigure the service when nodes become unavailable or are added to the cluster.

A client emulator is used to exercise the service. The workload consists of a number of concurrent clients that repeatedly open sessions with the service. Each client issues a request, receives and parses the reply, “thinks” for a while, and follows a link contained in the reply. A user-defined Markov model determines which link to follow. During our experiments, the overall load imposed on the system is 200 requests/second, which is approximately 35% of the service’s maximum achievable throughput. The code for the service and client emulator is publicly available from the DynaServer project [21] at Rice University.

Another important component of our experimental setup is a monitoring infrastructure that includes a shell that records and timestamps every single command (and the corresponding result) executed by the operator. The infrastructure also measures the system throughput on-the-fly, presenting it to the operator so that he/she can visually assess the impact of his/her actions on the system performance.

3.2 Experiments with operators

Our experiments can be categorized as either scheduled maintenance tasks or diagnose-and-repair tasks. Table 1 summarizes the classes of experiments.

Before having the operator interact with the system, we provide him/her with conceptual information on the system architecture, design, and interface. We convey this information verbally and through a graphical representation of the system which can be consulted at any time during an experiment. We also give the operator two

Task Category	Subcategory
Scheduled maintenance	Node addition
	Data migration
	Software upgrade
Diagnose-and-repair	Software misconfiguration
	Application crash/hang
	Hardware fault

Table 1: *Categories of experiments.*

sets of written instructions: general directions concerning the system interface and specific instructions about the task the operator will be doing. The operator is allowed to refer to both sets during the experiment.

A total of 21 people with varying skill levels volunteered to act as operators in our experiments: 14 graduate students, 2 operations staff members, and 5 professional programmers. The students and staff are from our own department; one of the staff members is the system administrator for one of our large clusters, and the other is the database administrator of our department. Two of the programmers used to work as system administrators, and four of them currently work for the Ask Jeeves commercial search engine.

To investigate the distribution of operator mistakes across different skill levels, we divide our operators into three categories: novice, intermediate, and expert. We deem the staff members and three of the professional programmers to be experts based on their experience in system administration. The remaining two programmers were classified as intermediate. Finally, we asked the graduate students to complete a questionnaire designed to assess their operation experience. Based on their responses, we ended up with five experts, five intermediates, and eleven novices. In Section 3.9, we discuss the breakdown of mistakes across these operator categories.

We gave the novice operators a “warm up” task involving the addition of a new Web server to the system to give them the opportunity to understand the system configuration and tier-to-tier communication issues, as well as crystallize the information we had conveyed orally. For this task, we provided very detailed instructions. (In our study, we do not take the mistakes made in this warm up task into consideration.)

All sets of instructions, the questionnaire, and the operator behavior data we collected are available at <http://vivo.cs.rutgers.edu/>.

3.3 Maintenance task 1: add an application server

In this experiment we ask the operator to add a Tomcat server to the second tier. In a nutshell, the operator is supposed to copy the Tomcat binary distribution from

any machine in the second tier to the specified new machine and configure it properly, so that it can exchange information with the database in the third tier. In addition, it is necessary to correctly reconfigure and restart the Web servers for the newly added Tomcat server to actually receive and process requests. The experiment is set up in such a way that the system has enough resources to handle the load imposed by the client emulator; hence, the new Tomcat server does not imply any increase in throughput.

This experiment has been conducted with eight novice operators, four intermediates, and two experts, with an average time per run of one hour. Two of the operators were completely successful in that they did not make any configuration mistakes or affect the system’s ability to service client requests more than what was strictly necessary. On the other hand, the other operators made mistakes with varying degrees of severity. The next few paragraphs discuss these mistakes.

Apache misconfigured. This was the most common mistake. We recognized four different flavors of it, all of them affecting the system differently. In the least severe misconfiguration, three novice operators did not make all the needed modifications to the Apache configuration file. In particular, they added information about the new machine to the file, but forgot to add the machine’s name to the very last line of file, which specifies the Tomcat server names. As a result, even though Tomcat was correctly started on the new machine, client requests were never forwarded to it. The operators who made this mistake either did not spend any time looking at the Apache log files to make sure that the new Tomcat server was processing requests or they analyzed the wrong log files. Although this misconfiguration did not affect the system performance immediately, it introduced a latent error.

Another flavor of Apache misconfiguration was more subtle and severe in terms of performance impact. One novice operator introduced a syntax error when editing the Apache configuration file that caused the module responsible for forwarding requests to the second tier (`mod_jk`) to crash. The outcome was the system’s inability to forward requests to the second tier. The operator noticed the problem by looking desperately at our performance monitoring tool, but could not find the cause after 10 minutes trying to do so. At that point, he gave up and we told him what the problem was.

One more mistake occurred when one expert and one novice modified the configuration file but left two identical application server names. This also made `mod_jk` crash and led to a severe throughput drop: a decrease of about 50% when the mistake affected one of the Web servers, and about 90% when both Web servers were compromised. The operators who made this mistake were not able to correct the problem for 20 minutes on

average. After this period, they decided not to continue and we showed them their mistakes.

Finally, one intermediate operator forgot to modify the Apache configuration file altogether to reflect the addition of the new application server. This mistake resulted in the inability of Apache to forward requests to the new application server. The operator was able to detect his mistake and fix the problem in 24 minutes.

Apache incorrectly restarted. In this case, one intermediate operator reconfigured one Apache distribution and launched the executable file from another (there were two Apache distributions installed on the first-tier machines). This mistake made the affected Web server become unable to process any client requests.

Bringing down both Web servers. In a few experiments, while reconfiguring Apache to take into account the new application server, two novice and three intermediate operators unnecessarily shutdown both Web servers at the same time and, as a consequence, made the whole service unavailable.

Tomcat incorrectly started. One novice and one expert were unable to start Tomcat correctly. In particular, they forgot to obtain root privileges before starting Tomcat. The expert operator started Tomcat multiple times without killing processes remaining from the previous launches. This scenario led to Tomcat silently dying. To make matters worse, since the heartbeat service — which is a separate process — was still running, the Web servers continued forwarding requests to a machine unable to process them. The result was substantially degraded throughput. The operators corrected this mistake in 22 minutes on average.

3.4 Maintenance task 2: upgrade the database machine

The purpose of this experiment is to migrate the MySQL database from a slow machine to a powerful one, which is equipped with more memory, a faster disk, and a faster CPU. (Note that we use the fast machine in all other experiments.) Because the database machine is the bottleneck of our testbed and the system is saturated when the slow machine is used, the expected outcome of this experiment is higher service throughput.

This experiment involves several steps: (1) compile and install MySQL on the new machine; (2) bring the whole service down; (3) dump the database tables from the old MySQL installation and copy them to the new machine; (4) configure MySQL properly by modifying the `my.cnf` file; (5) initialize MySQL and create an empty database; (6) import the dumped files into the empty database; (7) modify the relevant configuration files in all application servers so that Tomcat can forward

requests to the new database machine; and (8) start up MySQL, all application servers, and Web servers.

Four novices, two intermediates, and two experts performed this task; the average time per run was 2 hours and 20 minutes. We next detail the mistakes observed.

No password set up for MySQL root user. One novice operator failed to assign a password to the MySQL root user, during MySQL configuration. This mistake led to a severe security vulnerability, allowing virtually anyone to execute any operation on the database.

MySQL user not given necessary privileges. As part of the database migration, the operators need to ensure that the application servers are able to connect to the database and issue the appropriate requests. This involves reconfiguring Tomcat to forward requests to the new database machine and granting the proper privileges to the MySQL user that Tomcat uses to connect to the database. One novice and one expert did not grant the necessary privileges, preventing all application servers from establishing connections to the database. As a result, all Tomcat threads eventually got blocked and the whole system became unavailable. The expert managed to detect and correct the problem in 45 minutes. The novice did not even try to identify the problem.

Apache incorrectly restarted. One intermediate operator launched Apache from the wrong distribution, while restarting the service. Again, this mistake caused the service to become completely unavailable. It took the operator 10 minutes to detect and fix the mistake.

Database installed on the wrong disk. The powerful machine had two disks: a 15K RPM SCSI disk and a 7200 RPM IDE disk. Given that the database machine was known to be the bottleneck of our system and database migration was needed so that the service could keep up with the load imposed by the emulated clients, the operators should not have hesitated to install MySQL on the faster SCSI disk. One novice operator installed the database on the slow disk, limiting the throughput that can be achieved by the service. The operator never realized his mistake.

3.5 Maintenance task 3: upgrade one Web server

In this experiment, the operators are required to upgrade Apache from version 1.3.27 to version 2.0.49 on one machine. In a nutshell, this involves downloading the Apache source code from the Web, compiling it, configuring it properly, and integrating it into the service.

Two intermediate and three expert operators participated in this maintenance experiment. The average time per run was about 2 hours. We describe the observed mistakes next.

Apache misconfigured. Before spawning the Web server processes, Apache version 2.0.49 automatically invokes a syntax checker that analyzes the main configuration file to make sure that the server will not be started in the event of configuration syntax errors. This feature is extremely useful to avoid exposing operator mistakes to the live system. In our experiments, the syntax checker actually caught three configuration mistakes involving the use of directives no longer valid in the newer Apache version. However, as the checker is solely concerned with syntax, it did not catch some other configuration mistakes. One expert launched Apache without specifying in the main configuration file how Apache should map a given URL to a request for a Tomcat servlet. This misconfiguration led to the inability of the upgraded Apache to forward requests to the second tier, causing degraded throughput. The operator fixed this problem after 10 minutes of investigation.

In addition, a latent error resulted from two other misconfigurations. Two experts and one intermediate configured the new Apache server to get the HTML files from the old Apache's directory tree. A similar mistake was made with respect to the location of the heartbeat service program. The latent error would be activated if someone removed the files belonging to the old distribution.

Yet another mistake occurred when one expert correctly realized that the heartbeat program should be executed from the new Apache's directory tree, but incorrectly specified the path for the program. In our setup, the heartbeat program is launched by Apache. Because of the wrong path, `mod_jk` crashed when the new Apache was started. This made the new server unable to process requests for dynamic content, resulting in throughput degradation. The operator was able to fix this problem in 13 minutes.

3.6 Diagnose-and-repair task 1: Web server misconfiguration and crash

To observe the operator behavior resulting from latent errors that become activated, we performed experiments in which an Apache server misconfiguration and later crash are injected into the system. This sequence of events mimics an accidental misconfiguration or corruption of the configuration file that is followed by the need to restart the server.

In more detail, the system starts operating normally. At some point after the experiment has begun, we modify the configuration file pertaining to `mod_jk` in one of the Apache servers, so that a restart of the server will cause a segmentation fault. Later, we crash the same server to force the operator to restart it. As soon as the server is abnormally terminated, the throughput decreases to half of its prior value. The operators' task is to diagnose and fix

the problem, so that normal throughput can be regained.

This experiment was presented to three novices, three intermediates, and two experts, and the average time per run was 1 hour and 20 minutes. Two operators were able to both understand the system's malfunctioning and fix it in about 1 hour and 15 minutes. All operators — even the successful ones — made some mistakes, most of which aggravated the problem. All of the mistakes were caused by misdiagnosing the source of the service malfunction.

Misdiagnosis. Due to misdiagnosis, operators of all categories unnecessarily modified configuration files all over the system, which, in one case, caused the throughput to drop to zero. The apparent reason for such behavior was the fact that some operators were tempted to literally interpret the error messages appearing in the log files, instead of reasoning about what the real problem was. In other words, when reading something like "Apache seems to be busy; you should increase the *MaxClients* parameter..." in a log file, some operators performed the suggested action without further reasoning.

We also noticed the mistake of starting the wrong Apache distribution (as previously discussed) made by one novice and one intermediate, severely degrading the throughput or making it drop to zero. A couple of operators even suggested the replacement of hardware components, as a result of incorrectly diagnosing the problem as a disk or a memory fault.

3.7 Diagnose-and-repair task 2: application server hang

In this experiment, we inject another kind of fault: we force Tomcat to hang on three second-tier machines. The system is working perfectly until we perturb it.

We conducted this experiment with two novices, one intermediate, and one expert operator. All operators were able to detect the fault and fix the problem after 1 hour and 30 minutes on average. However, we noticed some mistakes as discussed next.

Tomcat incorrectly restarted. One novice operator restarted one of the two working servlet servers without root privileges, causing it to crash. This caused the service to lose yet another servlet server and the remaining one became overloaded. The operator was only able to detect the crashed Tomcat server 20 minutes later.

Database unnecessarily restarted. While trying to diagnose the problem, one novice operator unnecessarily restarted the database server. As the database machine is not replicated, bringing it down results in the system's inability to process most requests.

MySQL denied write privileges. One intermediate operator, while trying to diagnose the problem, decided to thoroughly verify the MySQL files. The operator in-

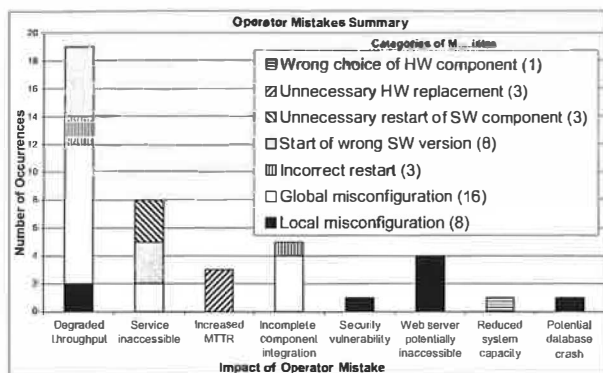


Figure 1: Operator mistakes and their impact.

advertently write-protected the whole MySQL directory tree, leading to MySQL's inability to write to the tables. This mistake did not cause any immediate effect because the files containing the tables had already been opened by the database server. However, this mistake led to a latent error that would arise if, for some reason, the database had to be restarted.

3.8 Diagnose-and-repair task 3: disk fault in the database machine

In this experiment, we use the Mendokusis fault-injection and network-emulation tool [15] to force a disk timeout to occur periodically in the database machine. The timeouts are injected according to an exponential inter-arrival distribution with an average rate of 0.03 occurrences per second. Since the database is the bottleneck, the disk timeouts substantially decrease the service throughput.

Four experts participated in this experiment. Of these four operators, three were unable to discover the problem after interacting with the system for 2 hours on average, and the other one correctly diagnosed the fault in 34 minutes. Throughout their interaction with the service, the unsuccessful operators made mistakes caused by misdiagnosing the real root of the problem.

Misdiagnosis. Two operators ended up diagnosing the fault as an “intermittent network problem” between the second and third tiers. Before the operators reached that conclusion, we had observed other incorrect diagnoses on their part such as DoS attack, Tomcat misconfiguration, and lack of communication between the first and second tiers. The other operator was suspicious of a MySQL misconfiguration and tried to adjust some parameters of the database and subsequently restarted it.

Under the influence of error messages reported in the log files, one operator changed, in two application server machines, the port which Tomcat was using to receive requests from Apache; as a result, the affected application servers became unreachable. The other two operators

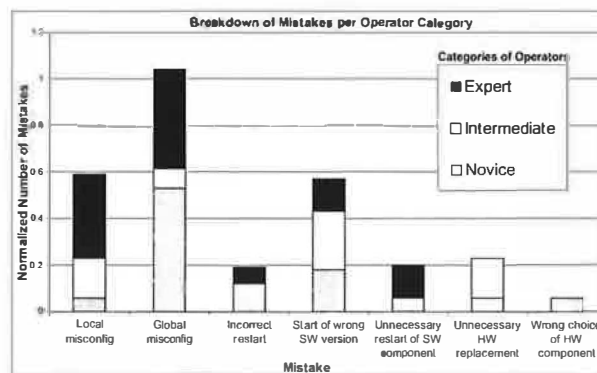


Figure 2: Operator mistakes per operator category.

looked at the main kernel log on the database machine and saw several messages logged by the SCSI driver reporting the disk malfunction. Unfortunately, they ignored such messages and did not even suspect that the disk was misbehaving.

3.9 Summary

Figures 1 and 2 summarize our findings. The X-axis in Figure 1 indicates the effects of the different operator mistakes, whereas the stacked bars show the number of occurrences of the mistake categories listed in the legend. The legend also shows the number of mistakes in each mistake category. In the figure, “incomplete component integration” refers to scenarios in which an added component is not seen by other components, “wrong choice of HW component” refers to installing the database on a slow disk, “unnecessary HW replacements” refers to misdiagnosing service malfunction as a hardware problem, and “unnecessary restart of SW component” refers to restarts of the database server. Overall, we observed 42 mistakes. In some cases, a single operator made more than one mistake during a particular experiment.

As we indicated before, misconfiguration was the most frequent mistake in our experiments. In Figure 1, we distinguish between local and global misconfiguration mistakes. Global misconfiguration refers to inconsistencies in one or more configuration files compromising the communication between system components, whereas local misconfiguration refers to misconfigurations that affect only one component of the system.

A local misconfiguration is a configuration mistake that caused Tomcat to crash, led to a security vulnerability, or could potentially prevent Apache from servicing requests. Global misconfigurations involve mistakes that: (1) prevented one or both Web servers from either forwarding requests to any application server machine, or sending requests to a newly added application server; (2) prevented the application servers from establishing

connections with the database server; (3) prevented one or both Web servers from processing requests; and (4) led one or both Web servers to forward requests to a non-existing application server.

In Figure 2, we show the distribution of mistakes across our three operator categories. Given that no operator took part in all types of experiments, we normalized the number of mistakes by dividing it by the total number of experiments in which the corresponding operator category participated. As the figure illustrates, experts made mistakes (especially misconfigurations) in a significant fraction of their experiments. The reason for this counter-intuitive result is that the hardest experiments were performed mostly by the experts, and those experiments were susceptible to local and global misconfiguration.

As we discuss in detail later, our validation approach would have been able to catch the majority (66% or 28 mistakes) of the 42 mistakes we observed. The remaining 14 mistakes, including unnecessary software restarts and unnecessary hardware replacements, were made by expert (6 mistakes), intermediate (4 mistakes), and novice (4 mistakes) operators.

4 Validation

Given that even expert operators make many mistakes, we propose that operator actions should be *validated* before their effects are exposed to end users. Specifically, we build a validation infrastructure that allows components to be validated in a slice of the online system itself, rather than being tested in a separate offline environment.

We start this section with an overview of our proposed validation approach. Then, we describe a prototype implementation and our experience in modifying the three-tier auction service to include validation. We have also implemented a validation framework for the PRESS clustered Web server [7]. PRESS is an interesting counter-point to our multithreaded auction service as it is a single-tier, event-based server. Our earlier technical report [18] discusses this implementation. Finally, we close the section with a discussion of how operators can still make mistakes even with validation and with a comparison of validation with offline testing and undo.

4.1 Overview

A validation environment should be closely tied to the online system for three reasons: (1) to avoid latent errors that escape detection during validation but become activated in the online system, because of differences between the validation and online environments; (2) to load components under validation with as realistic a workload as possible; and (3) to enable operators to bring

validated components online without having to change any of the components' configurations, thereby minimizing the chance of new operator mistakes. On the other hand, the components under validation, which we shall call *masked* components for simplicity, must be *isolated* from the live service so that incorrect behaviors cannot cause service failures.

To meet the above goals, we divide the cluster hosting a service into two logical slices: an online slice that hosts the live service and a validation slice where components can be validated before being integrated into the live service. Figure 3 shows this validation architecture in the context of the three-tier auction service. To protect the integrity of the live service without completely separating the two slices (which would reduce the validation slice to an offline testing system), we erect an isolation barrier between the slices but introduce a set of connecting *shunts*. The shunts are one-way portals that duplicate requests and replies (i.e., inputs and outputs) passing through the interfaces of the components in the live service. Shunts either log these requests and replies or forward them to the validation slice. Shunts can be easily implemented for either open or proprietary software, as long as the components' interfaces are well-defined.

We then build a validation harness consisting of *proxy components* that can be used to form a virtual service around the masked components as shown by the dashed box in Figure 3. Together, the virtual service and the duplication of requests and replies via the shunts allow operators to validate masked components under realistic workloads. In particular, the virtual service either replays previously recorded logs or accepts forwarded duplicates of live requests and responses from the shunts, feeds appropriate requests to the masked components, and verifies that the outputs of the masked components meet certain validation criteria. Proxies can be implemented by modifying open source components or wrapping code around proprietary software with well-defined interfaces.

Finally, the validation harness uses a set of *comparator functions* to test the correctness of the masked components. These functions compute whether some set of observations of the validation service match a set of criteria. For example, in Figure 3, a comparator function might determine if the streams of requests and replies going across the pair of connections labeled A and those labeled B are similar enough (A to A and B to B) to declare the masked Web server as working correctly. If any comparison fails, an error is signaled and the validation fails. If after a threshold period of time all comparisons match, the component is considered validated.

Given the above infrastructure, our approach is conceptually simple. First, the operator places the components to be worked on in the validation environment, effectively masking them from the live service. The oper-

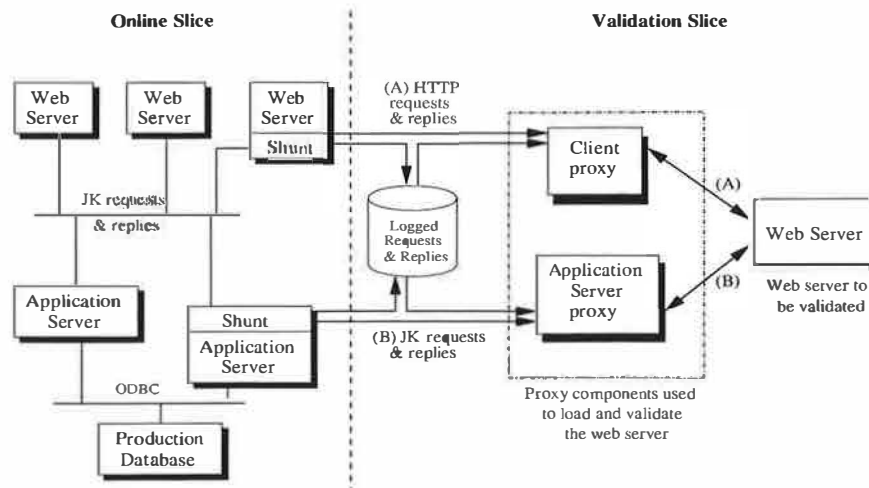


Figure 3: The three-tier auction service with validation. In this particular case, a single component, a Web server, is being validated inside the validation slice. The validation harness uses one or more client proxies to load the Web server and one or more application server proxies to field requests for dynamic content from the Web server.

ator then acts on the masked components just as he/she would in the live service. Next, the operator instructs the validation harness to surround the masked components with a virtual service, load the components, and check their correctness. If the masked components pass this validation, the operator calls a migration function that fully integrates the components into the live service.

4.2 Slice isolation

A critical challenge in building a validation infrastructure is how to isolate the slices from each other yet allow validated components to be migrated to the live service without requiring any changes to their internal state and configuration parameters. Our current implementation achieves this isolation and transparent migration at the granularity of an entire node by running nodes over a virtual network created using Mendosus [15].

Given the virtual network, it is fairly easy to impose the needed isolation: Mendosus was designed to inject network faults that would partition a network. Thus, we simply instruct Mendosus to partition our system into two parts to isolate the two slices from each other. Mendosus runs on each node, and, when enforcing a network partition, drops all packets, including multicast packets, that would normally flow between nodes separated by the partition. This enforced partition means that nodes in the validation slice can see each other but not the ones in the online slice and vice-versa. (To tunnel through this barrier, the shunts forward information to special nodes that have the privilege to bypass the network partition.)

Our virtual network then allows a node to be migrated between the slices without requiring any changes to the node's network configurations, as long as the software

components comprising the service can dynamically discover each other and automatically adjust the service configuration to include all components running in a slice, which is a characteristic of all production-level services. This transparent network-level migration is particularly critical for detecting the global misconfiguration mistakes described in Section 3.9. Migrating live components without modifying their internal state is more difficult. We detail how we accomplish this migration below.

4.3 Validation strategies

An inherently difficult problem for validation is how to drive masked components with realistic workloads and validate their correctness. Consider the validation of a single component. One possible approach is to create a demanding workload to stress-test the component. Such an approach lends itself to *trace-based* techniques, where the requests and replies passing through the shunts of an equivalent live component are logged and later replayed. During the replay, the logged replies can be compared to the replies produced by the masked component. A second approach, *replica-based*, is to use the current offered load on the live service, where requests passing through the shunts of an equivalent live component are duplicated and forwarded in real-time to the validation harness to drive the masked component. The shunts also capture the replies generated by the live component and forward them to the harness, which compares them against the replies coming from the masked component.

The core differences between the two approaches are the assumptions about the request stream and the connections between components. For example, logged (or

even synthetic) request streams that exercise known difficult cases may be preferable to a light live load. On the other hand, replica-based validation may be necessary if the service or data sets have changed sufficiently that previously collected traces are no longer applicable.

In reality, these two strategies are not mutually exclusive; a particular validation might apply one or both approaches before concluding that a masked component is working properly. Further, we can use the same supporting infrastructure to implement both approaches: data collected by the shunts can either be saved to disk or forwarded directly to the validation slice. Thus, building both of these approaches in the same system is quite desirable, as long as there is sufficient storage bandwidth (trace collection) and network bandwidth (live forwarding). We explore the resource requirements of these approaches further in Section 5.1.

State management. An important issue related to the validation strategy is component state management. In our work, component state is any application-defined, externalizable data (either hard or soft), accrued during service execution that can affect subsequent responses. Specifically, our validation system faces two state management issues: (1) how to initialize a masked component with the appropriate internal state so that it can handle the validation workload correctly, and (2) how to migrate a validated component to the online slice without migrating state that may have accumulated during validation but is not valid for the live service.

The way we initialize the masked component depends on the validation strategy. In trace-based validation, a masked component is initialized with state recorded in the trace. We record this state before trace capture, but after we halt the component to be traced by temporarily buffering all incoming requests in the shunts and allowing all current requests to complete. In replica-based validation, the masked component is initialized with a copy of the state of the associated live component. We make the copy exactly as in trace-based validation, except that the state is forwarded to the masked component instead of being saved to disk.

In general, this strategy should be applicable to any component that supports the checkpointing of its state, including proprietary systems. However, care must be taken to avoid capturing state under overload conditions, when sufficient buffering may not exist to enable the momentary halting of the component. Further, it may be impossible to capture a large amount of state, such as that of a database, online. In this case, the component may need to be removed from active service before its state can be captured.

After validation, the operator can move a component holding only soft state to the online slice by restarting it and instructing Mendosus to migrate it to the online

slice, where it will join the service as a fresh instance of that component type. For components with hard state, migration to the online slice may take two forms: (1) if the application itself knows how to integrate a new component of that type into the service (which may involve data redistribution), the component can be migrated with no state similar to components with only soft state; (2) otherwise, the operator must restart the component with the appropriate state before migrating it.

Multi-component validation. While the above discussion assumes the validation of a single component for simplicity, in general we need to validate the interaction between multiple components to address many of the global configuration mistakes described in Section 3. For example, when adding a new application server to the auction service, the Web servers' list of application servers must expand to include the new server. If this list is not updated properly, requests will not be forwarded to the new server after a migration, introducing a latent error. To ensure that this connection has been configured correctly, we must validate the existing Web servers and the new application server concurrently.

We call the above multi-component validation and currently handle it as follows. Suppose the operator needs to introduce a new component that requires changes to the configurations of existing live components, such as the adding of an application server. The operator would first introduce the new component to the validation slice and validate its correctness as discussed above. Next, each component from the live service whose configuration must be modified (so that it can interact properly with this new component) is brought into the validation slice one-by-one and the component pair is validated together to ensure their correct interoperability. After this validation, the existing component is migrated back into the online slice; the new component is only migrated to the online slice after checking its interactions with each existing component whose configurations had to be changed.

Note that there are thus at most two components under validation at any point in time in the above multi-component validation approach. In general, multi-component validation could be extended to include up to k additional components, but so far we have not found it necessary to have $k > 1$.

4.4 Implementing validation

Setting up the validation infrastructure involves modifying the online slice to be able to shunt requests and responses, setting up the harness composed of proxies within the validation slice, defining appropriate comparators for checking correctness, and implementing mechanisms for correct migration of components across

the two slices. In this section, we first discuss these issues and then comment on their implementation effort with respect to the auction service.

Shunts. We have implemented shunts for each of the three component types in the three-tier auction service. This implementation was straightforward because all inter-component interactions were performed using well-defined middleware libraries. Figure 3 shows that client requests and the corresponding responses are intercepted within the JK module on the Apache side. The current implementation does not intercept requests to static content, which represent a small percentage of all requests. The requests and responses to and from the database, via the Open DataBase Connectivity (ODBC) protocol, are intercepted in Tomcat's MySQL driver as SQL queries and their corresponding responses. In addition to duplicating requests and replies, we also tag each request/reply pair with a unique identifier. This ID is used by the validation harness to identify matching requests and responses generated by a masked component with the appropriate logged or forwarded requests and responses from the live system to which the comparator functions can be applied.

Validation harness. The validation harness needs to implement a service around the masked components in order to exercise them and check their functionalities. For example, to validate an application server in the auction service, the validation harness would need to provide at least one Web server and one database server to which the masked application server can connect.

One approach to building a service surrounding a masked component is to use a complete set of real (as opposed to proxy) components. This is reminiscent of offline testing, where a complete model of the live system is built. Although this approach is straightforward, it has several disadvantages. First, we would need to devote sufficient resources to host the entire service in the validation slice. Second, we would need a checkpoint of all service state, such as the content of the database and session state, at the beginning of the validation process. Finally, we would still need to fit the appropriate comparators into the real components.

To address the above limitations, we built lighter weight component proxies that interact with the masked component without requiring the full service. The proxies send requests to the masked component and check the replies coming from it. For services in which communicating components are connected using a common connection mechanism, such as event queues [7, 22], it is straightforward to realize the entire virtual service as collection of such queues in a single proxy. For heterogeneous systems like the auction service however, the tiers connect to each other using a variety of communication

mechanisms. Thus, we have built different proxies representing the appropriate connection protocols around a common core.

The auction service required four different proxies, namely client, Web server, application server, and database proxies. Each proxy typically implements three modules: a membership protocol, a service interface, and a messaging core. The membership protocol is necessary to guarantee dynamic discovery of nodes in the validation slice. The service interface is necessary for correct communication with interacting components. The common messaging core takes shunted or logged requests to load the masked components and responds to requests made by the masked components.

Regarding state management, we currently focus solely on the soft state stored (in main memory) by application servers, namely the auctions of interest to users. To handle this state, we extend the application servers to implement an explicit state checkpointing and initialization API. This API is invoked by the proxies to initialize the state of the masked application server with that of an equivalent live component.

Our experience indicates that the effort involved in implementing proxies is small and the core components are easily adaptable across services. Except for the messaging core, which is common across all proxies, the proxies for the auction service were derived by adding/modifying 232, 307, and 274 non-comment source lines (NCSL) to the Rice client emulator [21], the Apache Web server, and the Tomcat application server, respectively. The NCSL of the application server also includes the code to implement the state management API. The MySQL database proxy was written from scratch and required only 384 NCSL.

Comparator functions. Our current set of comparator functions includes a throughput-based function, a flow-based function, and a set of component-level data matching functions. The throughput-based function validates that the average throughput of the masked component is within the threshold of an expected value. The flow-based function ensures that requests and replies indeed flow across inter-component connections that are expected to be active. Finally, the data matching functions match the actual contents of requests and replies. Due to space limitations, we only consider this last type of comparator function in this paper. We describe below how we handle the cases where exact matches are not possible because of non-determinism.

Non-determinism. Non-determinism can take several forms: in the timing of responses, in the routing of requests, and in the actual content of responses. We found that timing and content non-determinism were not significant problems in the applications we studied. On

the other hand, because data and components are replicated, we found significant non-determinism in the routing of requests. For example, in the auction service, a Web server can forward the first request of a session to any of the application servers. In PRESS, routing non-determinism can lead to a local memory access, sending the request to a remote node, or to the local disk. The proxies need to detect that these behaviors are equivalent and possibly generate the appropriate replies. Fortunately, implementing this detection and generation was quite simple for both our services.

4.5 Example validation scenario

To see how the above pieces fit together, consider the example of an operator who needs to upgrade the operating system of a Web server node in the auction service (Figure 3). The operator would first instruct Mendosus to remove the node from the online slice, effectively taking it offline for the upgrade. Once the upgrade is done and the node has been tested offline, e.g. it boots and starts the Web server correctly, the operator would instruct Mendosus to migrate the node to the validation slice. Next, the validation harness would automatically start an application server proxy to which the Web server can connect. Once both components have been started, they will discover each other through a multicast-based discovery protocol and interconnect to form a virtual service. The harness would also start a client proxy to load the virtual service. Under trace-based validation, the harness would then replay the trace, with the client proxy generating the logged requests and accepting the corresponding replies (shown as A in Figure 3), and the application server proxy accepting requests for dynamic content from the Web server and generating the appropriate replies from logged data (shown as B). The harness would also compare all messages from the Web server to the application server and client proxies against the logged data. Once the trace completes without encountering a comparison mismatch, the harness would declare the Web server node validated. Finally, the operator would place the node back into the live service by restarting it and instructing Mendosus to migrate it to the online slice *without further changing any of its configurations*.

4.6 Discussion

Having described validation in detail, we now discuss the generality and limitations of our prototype, and some remaining open issues. We also compare our approach to offline testing and undo.

Generality. While our implementations have been done only in the context of two systems, the auction service

and the PRESS server, we believe that much of our infrastructure is quite general and reusable. First, the auction service is implemented by three widely used servers: Apache, Tomcat, and MySQL. Thus, the proxies and shunts that we have implemented for the auction service should be directly reusable in any service built around these servers. Even for services built around different components, our shunts should be reusable as they were implemented based on standard communication libraries. Further, as already mentioned, implementing the proxies requires relatively little effort given a core logging/forwarding and replay infrastructure. Finally, our experience with PRESS suggests that event-based servers are quite amenable to our validation approach, as all interactions between components pass through a common queuing framework.

Perhaps the most application-specific parts of our validation approach are the comparator functions and the state management API. Generic comparator functions that check characteristics such as throughput should be reusable. However, comparator functions that depend on the semantics of the applications are unavoidably application-specific and so will likely have to be tailored to each specific application. The state management API is often provided by off-the-shelf stateful servers; when those servers do not provide the API, it has to be implemented from scratch as we did for Tomcat.

Limitations. The behavior of a component cannot be validated against a known correct instance or trace when the operator actions properly change the component's behavior. For example, changes to the content being served by a Web server correctly leads to changes in its responses to client requests. However, validation is still applicable, as the validation harness can check for problems such as missing content, unexpected structure, etc. In addition, once an instance has been validated in this manner, it can be used as a reference point for validating additional instances of the same component type. Although this approach introduces scope for mistakes, we view this as an unavoidable bootstrapping issue.

Another action that can lead to mistakes is the restart of components in the live service after validation, a step that is typically necessary to ensure that the validated components will join the live service with the proper state. However, the potential for mistakes can be minimized by scripting the restart.

Open issues. We leave the questions of what exactly should be validated, the degree of validation, and for how long as open issues. In terms of trace-based validation, there are also the issues of when traces should be gathered, how often they should be gathered, and how long they should be. All of these issues boil down to policy decisions that involve trade-offs between the probabil-

ity of catching mistakes vs. the cost of having resources under validation rather than online. In our live operator experiments with validation, we leave these decisions to the discretion of the operator. In the future, we plan to address these issues in more detail.

One interesting direction is to study strategies for dynamically determining how long validation should take based on the intensity of the offered load. During periods of heavy load, validation may retain resources that could be used more productively by the live service.

We also plan to explore a richer space of comparator functions. For example, weaker forms of comparison, such as using statistical sampling, may greatly improve performance while retaining the benefit of validation.

Comparison against offline testing. Sites have been using *offline testing* for many years [3]. The offline testing environment typically resembles the live service closely, allowing operators to act on components without exposing any mistakes or intermediate states to users. Once the components appear to be working correctly, they can be moved into the live service.

The critical difference between our validation approach and offline testing is the fact that our validation environment is an extension, rather than a replica, of the live service. Thus, misconfiguration mistakes can occur in offline testing when the software or hardware configurations have to be changed during the moving of the components to the live service. For example, adding an application server requires modifying the configuration file of all the Web servers. Although a misconfigured Web server in the offline environment can be detected using offline testing, failing to correctly modify the live configuration file would result in an error. Furthermore, other mistakes could be made during these actions and consequently be exposed to the end users.

In order to gauge the ability of offline testing to catch the mistakes that we have observed (Section 3), we assume that trivial mistakes that do not involve inter-component configuration are unlikely to be repeated in the live system. Under this assumption, offline testing would have allowed the operator to catch (1) all instances of the “start of wrong software version” category, (2) the instance of local misconfiguration that caused the database security vulnerability (assuming that the operators would explicitly test that case), and (3) some instances of global misconfiguration, such as the one in which the incorrect port was specified in the Tomcat configuration file. Overall, we find that offline testing would only have been able to deal with 17 out of the 42 mistakes that we observed, i.e. 40% of the mistakes, while our validation approach would have caught 66%.

Comparison against undo. Undo [6] is essentially orthogonal to our validation approach. Undo focuses on

enabling operators to fix their mistakes by bringing the service back to a correct state. The focus of validation is to hide operator actions from the live service until they have been validated in the validation environment. As such, one technique can benefit from the other. Undo can benefit from validation to avoid exposing operator mistakes to the live service, and thus the clients, whereas validation can benefit from undo to help correct operator mistakes in the validation environment.

Assuming that undo would be used alone, all mistakes we observed would have been immediately exposed to clients (either as an explicit error reply or as degraded server performance), except for the ones that caused latent errors and vulnerabilities. Nevertheless, undo would be useful in restoring the system to a consistent state after a malicious attack resulting from the database security vulnerability problem. If combined with offline testing, undo would have helped fix the mistakes detected offline.

5 Experimental Validation Results

In this section we first describe the performance impact of our validation infrastructure on the live service using micro-benchmarks. We then concretely evaluate our validation approach using several methods.

5.1 Shunting overheads

We measured the shunting overhead in terms of CPU usage, disk, and network bandwidth for interception, logging, and forwarding of inter-component interactions in the auction service. Note that while the comparator functions do not run on the online slice, the amount of information that we have to log or forward depends on the nature of the comparator functions we wish to use for validation. Thus, we investigate the performance impact of several levels of data collection, including collecting complete requests and replies, collecting only summaries, and sampling.

Figure 4 shows percentage CPU utilization for a live Web server in the auction service, as a function of the offered load. The utilization vs. load curve for the unmodified server is labeled *base*. The curves for complete logging (used for trace-based validation) and forwarding (used for replica-based validation) are labeled *trace-val* and *replica-val*, respectively. These curves show the performance of shunting all requests and replies from both the Web and application servers. We can observe that complete logging causes an additional 24-32% CPU utilization for the throughput range we consider, whereas forwarding adds 29-39%.

A straightforward approach to reducing these overheads is to use only a summary of the responses. The *trace-summary-val* and *replica-summary-val* curves give

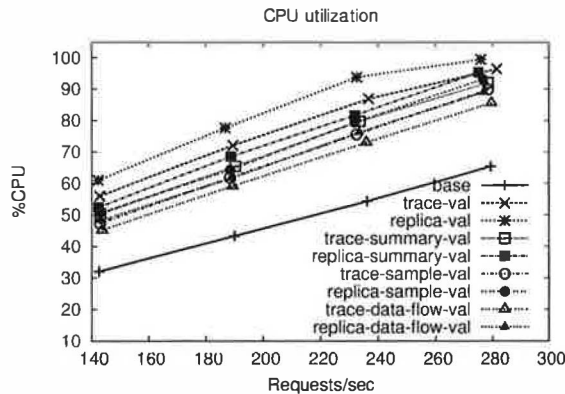


Figure 4: Processor overhead incurred in performing various validation operations on Web server.

the utilization for logging and forwarding, respectively, when we use the first 64 bytes of the HTTP responses. The additional CPU utilizations in this case for logging and forwarding are 18-25% and 20-27%, respectively.

A second approach to reducing the overheads is to sample, instead of collecting all requests and responses. To measure the impact of this approach, we programmed the shunts to log or forward only 50% of the client sessions, leading to the *trace-sample-val* and *replica-sample-val* curves. (This optimization was carefully implemented to avoid skewing the states of the compared components in replica-based validation.) The optimization reduces the overheads of logging and forwarding to 15-21% and 18-25%, respectively. Another sampling approach is to only shunt and compare the final input and outputs, ignoring internal messages. The *trace-data-flow-val* and *replica-data-flow-val* versions only sample HTTP requests and responses and ignore the JK messages. This approach leads to a CPU overhead of 13-19% and 16-22% for logging and forwarding, respectively.

We also examined the impact of shunting on disk and network bandwidth. We find that in the worst case, a bandwidth load of about 4 MB/s was generated. Using the every-other-session sampling method reduced the bandwidth load to about 2.5 MB/s, and the final-result-only sampling method further reduced it to 1.5 MB/s. These bandwidth results are encouraging, as they show that validation is unlikely to have an impact on throughput when using Gigabit networks and storage systems.

Overall, we find the CPU overheads to be significant for our base prototype, in many cases increasing utilization by 24%-39%, while the additional network and disk traffic was not significant. With different optimizations, we can reduce the CPU overheads to 13%-22%. These results are positive, given that *our approach loads only one or at most a few of the live components simultaneously, and only during validation*. Furthermore, since

many services run at fairly low CPU utilization (e.g., 50%-60%) to be able to deal with load spikes, this overhead should not affect throughputs in practice.

5.2 Buffering overheads

State checkpointing and initialization are performed by the shunts and proxies involved in the validation of a stateful server. We make these operations atomic by first draining all requests currently being processed by the components involved in the validation. After those requests complete, we start the required state operations. During the draining and the processing of the state operations, we buffer all requests arriving at the affected components. How long we need to buffer requests determines the delay imposed on (a fraction of) the requests and the buffer space overheads.

While the delays and space overheads can vary depending on size of the state and the maximum duration of an outstanding request, we find them to be quite tolerable for the validation of an application server in our auction service. In particular, we find that replica-based validation causes the longest buffering duration. However, even this duration was always shorter than 1 second, translating into a required buffer capacity of less than 150 requests for a heavily loaded replica server.

Since the average state size is small (less than 512 bytes) in the auction service, we synthetically increased the size of each session object up to 64 KBytes to study the impact of large states. This resulted in an overall response time of less than 5 seconds, which though not insignificant, is still manageable by the validation slice.

5.3 Operator mistake experiments

We used three different experimentation techniques to test the efficacy of our validation techniques. The experiments span a range of realism and repeatability. Our live-operator experiments are the most realistic, but are the least repeatable. For more repeatable experiments, we used operator emulation and mistake injection. For all experiments, the set up was identical to that described in Section 3, but we added two nodes to implement our validation infrastructure.

Live-operator experiments. For these experiments, the operator was instructed to perform a task using the following steps. First, the component that must be touched by the operator is identified and taken offline. Second, the required actions are performed. Next, the operator can use the validation slice to validate the component. The operator is allowed to choose the duration of the validation run. Finally, the operator must migrate the component to the online slice. Optionally, the operator can

place the component online without validation, if he/she is confident that the component is working correctly.

We ran eight experiments with live operators: three application server addition tasks (Section 3.3), three server upgrade tasks (Section 3.5), and one each of Web server misconfiguration (Section 3.6) and application server hang (see Section 3.7). Seven graduate students from our department acted as operators, none of whom had run the corresponding experiment without validation before.

We observed a total of nine operator mistakes in five of the experiments and validation was successful in catching six of them. Two of the mistakes not caught by validation were latent errors, whereas the other mistake, which led to an empty `htdocs` directory, was not caught only because our implementation currently does not test the static files served by the Web servers (as already mentioned in Section 4.4). Addressing this latter mistake merely requires an extension of our prototype to process requests to static files and their corresponding responses.

Interestingly, during one run of the Web server upgrade task, the operator started the new Apache without modifying the main configuration file, instead using the default one. Validation caught the mistake and prevented the unconfigured Apache from exposure. However, the operator tried to configure the upgraded Apache for 35 minutes; after a number of unsuccessful validations, he gave up. This example shows that another important area for future research is extending the validation infrastructure to help the operator more easily find the cause of an unsuccessful validation.

Operator-emulation experiments. In these experiments, a command trace from a previous run of an operator task is replayed using shell scripts to emulate the operator's actions. The motivation for this approach is that collection and reuse of operator's actions provides a repeatable testbed for techniques that deal with operator mistakes. This approach, however, has the limitation that once the operator's mistake is caught, subsequent recovery actions in the scripts are undefined. Nevertheless, we find the ability to repeat experiments extremely useful.

The traces were derived manually from the logs collected during the operator experiments described in Section 3. In the emulation scripts, an emulation step consists of a combination or summary of steps from the actual run with the goal of preserving the operator actions that impact the system. For example, if the operator performed a set of read-only diagnostic steps and subsequently modified a file, then the trace script will only perform the file modification.

We derived a total of 40 scripts from the 42 operator mistakes we observed; 2 mistakes were not reproducible due to infrastructure limitations. Table 2 summarizes our findings in terms of coverage, i.e., mistakes caught with respect to all mistakes. Validation was able to catch 26

Technique (40 total)	Coverage Impact	
	Immediate (29 total)	Latent (11 total)
Trace-based	22	0
Replica-based	22	0
Multi-component	22	4

Table 2: Coverage results of the emulation experiments.

of the 40 reproducible mistakes; 22 of these mistakes had an immediate impact while 4 caused latent errors. Both trace and replica-based validation caught all 22 mistakes causing an immediate impact. However, single-component validation failed to catch the latent errors during the addition of a new application server. These mistakes resulted in the Web servers not being updated correctly to include the new application server. These mistakes were caught using the multi-component validation approach described in Section 4.3.

Validation would have caught the 2 non-reproducible mistakes as well. These mistakes had an immediate impact similar to a number of the 22 reproducible mistakes caught by single-component validation. Assuming that these 2 mistakes would have been caught, our validation approach would detect a total of 28 out of the 42 mistakes (66%) we have observed.

Mistake-injection experiments. We hand-picked some additional mistakes and injected them to test the effectiveness of our validation system. Our goal is to see if our validation technique can cover mistakes that were not observed in the live-operator experiments.

To emulate mistakes in content management, we extended Mendosus to inject permission errors, missing files, and file-corruption errors. In PRESS, injection of permission and missing files errors were readily detected by our validation infrastructure. However, some file corruption errors were not caught because of thresholds in the comparator functions; typically a fraction of the bytes of a Web page are allowed to be different. While it is necessary to allow some slack in the comparator to prevent excessive false positives, this case illustrates that the comparator functions must be carefully designed to balance the false positive rate with exposing mistakes.

We also used Mendosus to perform manipulations of configuration parameters that only impacted the performance of the component. Specifically, we altered the in-memory cache size for PRESS and the maximum number of clients for Apache in the auction service. Both mistakes resulted in the component's performance dropping below the threshold of a throughput comparator and so were caught by validation. Once again, these experiments highlight the importance of designing suitable comparators and workloads.

6 Conclusions

In this paper, we collected and analyzed extensive data about operator actions and mistakes. From a total of 43 experiments with human operators and a three-tier auction service, we found 42 operator mistakes, the most common of which were software misconfiguration (24 mistakes) and incorrect software restarts (14 mistakes). A large number of mistakes (19) immediately degraded the service throughput.

Based on these results, we proposed that services should validate operator actions in a virtual environment before they are made visible to the rest of the system (and users). We designed and implemented a prototype of such a validation system. Our evaluation showed that the prototype imposes an acceptable performance overhead during validation. The results also showed that our prototype can detect 66% of the operator mistakes we observed.

Acknowledgments. We would like to thank our volunteer operators, especially the Ask Jeeves programmers and the DCS LCSR staff members, who donated considerable time for this effort. We would also like to thank Christine Hung, Neeraj Krishnan, and Brian Russell for their help in building part of the infrastructure used in our operator experiments. Last but not least, we would like to thank our shepherd, Margo Seltzer, for her extensive comments and support of our work.

References

- [1] S. Ajmani, B. Liskov, and L. Shriru. Scheduling and Simulation: How to Upgrade Distributed Systems. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, May 2003.
- [2] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: Real-Time Modelling and Performance-Aware Systems. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, May 2003.
- [3] R. Barrett, P. P. Maglio, E. Kandogan, and J. Bailey. Usable Autonomic Computing Systems: the Administrator's Perspective. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC'04)*, May 2004.
- [4] C. Boyapati, B. Liskov, L. Shriru, C.-H. Moh, and S. Richman. Lazy Modular Upgrades in Persistent Object Stores. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-oriented Programming, Languages, Systems and Applications (OOPSLA'03)*, Oct. 2003.
- [5] A. Brown. *A Recovery-oriented Approach to Dependable Services: Repairing Past Errors with System-wide Undo*. PhD thesis, Computer Science Division-University of California, Berkeley, 2003.
- [6] A. B. Brown and D. A. Patterson. Undo for Operators: Building an Undoable E-mail Store. In *Proceedings of the 2003 USENIX Annual Technical Conference*, June 2003.
- [7] E. V. Carrera and R. Bianchini. Efficiency vs. Portability in Cluster-Based Network Servers. In *Proceedings of the 8th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, June 2001.
- [8] M. Castro and B. Liskov. Proactive Recovery in a Byzantine-Fault-Tolerant System. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI'00)*, Oct. 2000.
- [9] M. Castro and B. Liskov. BASE: Using Abstraction to Improve Fault Tolerance. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, Oct. 2001.
- [10] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-Based Failure and Evolution Management. In *Proceedings of the International Symposium on Networked Systems Design and Implementation (NSDI'04)*, Mar. 2004.
- [11] J. Gray. Why do Computers Stop and What Can Be Done About It? In *Proceedings of 5th Symposium on Reliability in Distributed Software and Database Systems*, Jan. 1986.
- [12] J. Gray. Dependability in the Internet Era. Keynote presentation at the 2nd HDCC Workshop, May 2001.
- [13] Z. T. Kalbarczyk, R. K. Iyer, S. Bagchi, and K. Whisnant. Chameleon: A Software Infrastructure for Adaptive Fault Tolerance. *IEEE Transactions on Parallel and Distributed Systems*, 10(6), 1999.
- [14] E. Kiciman and Y.-M. Wang. Discovering Correctness Constraints for Self-Management of System Configuration. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC'04)*, May 2004.
- [15] X. Li, R. P. Martin, K. Nagaraja, T. D. Nguyen, and B. Zhang. Mendosus: SAN-Based Fault-Injection Test-Bed for the Construction of Highly Available Network Services. In *Proceedings of 1st Workshop on Novel Uses of System Area Networks (SAN-1)*, Jan. 2002.
- [16] D. E. Lowell, Y. Saito, and E. J. Samberg. Devirtualizable Virtual Machines - Enabling General, Single-Node, Online Maintenance. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, Oct. 2004.
- [17] B. Murphy and B. Levidow. Windows 2000 Dependability. Technical Report MSR-TR-2000-56, Microsoft Research, June 2000.
- [18] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and Dealing with Operator Mistakes in Internet Services. Technical Report DCS-TR-555, Department of Computer Science, Rutgers University, May 2004.
- [19] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why do Internet Services Fail, and What Can Be Done About It. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS'03)*, Mar. 2003.
- [20] D. A. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical Report UCB//CSD-02-1175, University of California, Berkeley, Mar. 2002.
- [21] Rice University. DynaServer Project. <http://www.cs.rice.edu/CS/Systems/DynaServer>, 2003.
- [22] M. Welsh, D. E. Culler, and E. A. Brewer. SED: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2001.

Configuration Debugging as Search: Finding the Needle in the Haystack

Andrew Whitaker, Richard S. Cox, and Steven D. Gribble

University of Washington

{andrew, rick, gribble}@cs.washington.edu

Abstract

This work addresses the problem of diagnosing configuration errors that cause a system to function incorrectly. For example, a change to the local firewall policy could cause a network-based application to malfunction. Our approach is based on searching across time for the instant the system transitioned into a failed state. Based on this information, a troubleshooter or administrator can deduce the cause of failure by comparing system state before and after the failure.

We present the Chronus tool, which automates the task of searching for a failure-inducing state change. Chronus takes as input a user-provided software probe, which differentiates between working and non-working states. Chronus performs “time travel” by booting a virtual machine off the system’s disk state as it existed at some point in the past. By using binary search, Chronus can find the fault point with effort that grows logarithmically with log size. We demonstrate that Chronus can diagnose a range of common configuration errors for both client-side and server-side applications, and that the performance overhead of the tool is not prohibitive.

1 Introduction

Continual change is a fact of life for software systems. For desktop machines, users can install new applications, apply software upgrades, change security policies, and alter system configuration options. Servers and other infrastructure services are also subject to frequent changes in functionality and administrative settings.

The ability to change is what gives software its vibrancy and relevance. At the same time, change has the potential to disrupt existing functionality. For example, software patches can break existing applications [5]. Seemingly unrelated applications can conflict — for example, by corrupting Windows registry keys or shared configuration options. Changes to security policies, while often necessary to respond to emerging threats, can disrupt functionality. For server-side applications, administrator actions and other “operator errors” [17] are a substantial contributor to overall downtime.

In most cases, these change-induced failures are

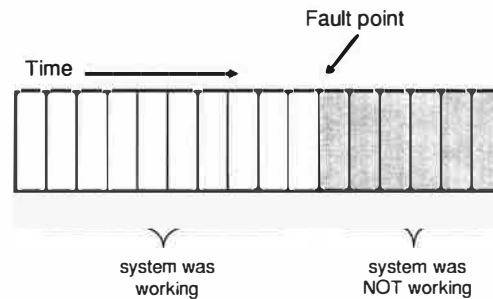


Figure 1: **Searching through time for a configuration error:** Chronus reveals configuration errors by pinpointing the instant in time the system transitioned to a failed state.

diagnosed by human experts such as system administrators. This approach suffers on a variety of fronts: trained experts are expensive, they are in short supply, and they are faced with escalating system complexity and change. In consequence, system administrative costs are approaching 60-80% of the total cost of ownership of information technology [12].

The goal of this work is to reduce the burden on human experts by partially *automating* problem diagnosis. In particular, we analyze the applicability of *search* techniques for diagnosing configuration errors. Our insight is that although computers cannot compete with human intuition, they are very effective at exploring a large configuration space. Our diagnosis tool, which we call Chronus, uses search to identify the specific time in the past when a system transitioned from a working to a non-working state, as shown in Figure 1. Using this information, an administrator can more easily diagnose why the system stopped working, for example, by comparing the file system state immediately before and after the fault point to determine the configuration change that “broke” the system.

1.1 Existing Approaches

In this work, we focus on automated problem diagnosis. For the sake of completeness, we briefly survey other approaches, arguing that the approach embodied by Chronus represents an advance for a significant class of configuration errors.

The best approach to dealing with configuration errors is **prevention**. Unfortunately, the complexity of today's systems makes it difficult to reason a priori about all possible side effects of a configuration change. One problem is that modern systems are built from components from many vendors, and there are few global mechanisms that are capable of understanding the effects of configuration changes in the large. The situation is further exacerbated by the inadequacy of analysis tools. For example, determining whether a software patch results in "equivalent" system behavior is intractable.

Recovery tools such as Windows XP Restore [24] create occasional state checkpoints, allowing users to "undo" [8] the effects of bad configuration changes. While effective in some situations, this approach faces several limitations. First, it requires the user to choose an appropriate state snapshot, which assumes that some form of problem diagnosis has already occurred. Second, recovery itself can corrupt system state, either by undoing "good" changes or restoring "bad" changes. Problem diagnosis in Chronus does not modify system state, and can therefore be safely employed in more situations.

Expert system diagnosis tools have a similar goal as Chronus, in that they attempt to map from symptoms to a root cause. A widely used (though rudimentary) example is the Windows "Help and Support Center." Expert systems typically rely on a static rule database, and are therefore only effective for known configuration errors. Arguably, known configuration errors would be better handled by improvements in software design or user interface. In addition, as systems grow more complex, static rule databases grow increasingly incomplete.

When all else fails, the last recourse is manual diagnosis by an expert. People have intuition and experience, letting them reason about unexpected situations. Unfortunately, human resources are scarce and costly, and mastering the complexity of today's software systems represents a significant hurdle to effective diagnosis.

1.2 The Chronus Approach

Chronus is a troubleshooting tool whose goals are to simplify the task of diagnosing a configuration error and to reduce the need for costly human expertise. Rather than requiring troubleshooters to answer the difficult question "why is the system not working," our tool instead requires them to supply a *software probe* (i.e., a script or program) that answers the simpler question "is the system currently working?" Given a probe, Chronus searches through time for the instant that the system transitioned from a working to a non-working state. As we will demonstrate, many common configuration errors can be diagnosed with simple shell scripts.

Chronus relies on several components. A *time-travel disk* [25] captures the progression of the system's durable

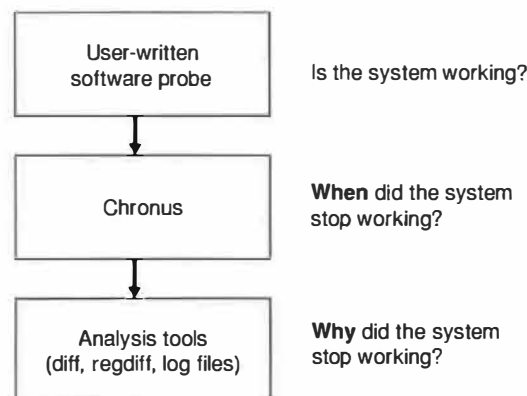


Figure 2: A Chronus debugging session: Given a user-supplied software probe, Chronus reveals when the system began failing. Based on this information, it is possible to understand the cause of failure using higher-level analysis tools.

state over time by logging disk block writes. Chronus uses the μ Denali virtual machine monitor [35] to *instantiate, boot, and test historical snapshots* of the system, including the complete operating system and application state. Chronus executes the user-supplied software probe to test whether a given historical state works correctly. Finally, Chronus relies on a *search strategy* to efficiently reduce the failure-inducing state change from a large sequence of historical states. In many cases, Chronus can use binary search, allowing for diagnosis time that scales logarithmically with log length.

The output from Chronus is the time of the fault point. Based on this timing information (the "when"), the troubleshooter can then use OS- or application-specific tools to diagnose the cause of the failure (the "why"). One simple but useful technique is to compare the complete file system state immediately before and after the failure using an invocation of the UNIX `diff` command. Figure 2 depicts the stages of a typical Chronus session.

1.3 Outline

In the remainder of this paper, we describe the design and implementation of Chronus, and we demonstrate its ability to help a troubleshooter diagnose significant configuration errors. The remainder of this paper is organized as follows. In Section 2, we describe some of the challenges we faced and design decisions that we made. Section 3 discusses the Chronus implementation. We evaluate Chronus in Section 4. After discussing related work in Section 6, we describe open problems and future work in Section 7, and we conclude in Section 8.

2 Challenges and Design Tradeoffs

In this section, we drill down into the major components of Chronus. In each case, we identify the major

challenges and describe the design tradeoffs we faced.

2.1 Time travel

Chronus relies on a time travel mechanism to instantiate previous system states. Traditional checkpointing systems capture the complete state of a system, including both persistent (e.g., disk contents) and transient state (e.g., memory and CPU state). This approach recreates previous states with high fidelity, but imposes a heavy overhead to continually flush memory state to disk. Approaches based on incremental logging (e.g., Revert [15]), reduce overhead during normal operation, but require more time to recreate a previous system state.

Instead of taking full checkpoints, Chronus only records updates to persistent storage. This allows for reasonable performance during both normal operation and problem diagnosis. As we demonstrate in Section 5.1, the overhead of our versioning storage system is primarily limited to disk space (which is plentiful) rather than degraded performance.

A drawback of disk-only state capture is that we sacrifice completeness: only errors that persist across system restarts are recorded by the time travel layer. Note, however, that some configuration changes require system restarts to take effect — for example, changes to shared libraries or the OS kernel typically require system reboots. For this type of “delayed release” configuration change, the on-disk state is more meaningful than the instantaneous characteristics of the running system.

2.1.1 Time-travel disks

Time-travel or versioning storage systems have been extensively studied. Proposed systems include versioning file systems [30, 31], source code repositories [14], time-travel databases [32], and the Peabody time-travel disk [25]. Taken as a whole, these systems demonstrate a tradeoff between completeness and high-level semantics (Figure 3). At one extreme, the time-travel disk offers the most completeness, in that it captures all state changes without requiring support from operating systems or applications. At the other extreme, relational databases offers strong data consistency semantics, but require applications to utilize a particular API.

For Chronus, we chose a storage system based on a time-travel disk. One of our goals was to avoid making assumptions about how and where configuration errors arise. Because of its low-level interface, a time-travel disk captures *all* local configuration changes, without regard to application or OS functionality. Chronus is to some degree “future-proof,” in that it can diagnose configuration errors for systems that have yet to be written.

A drawback of a time-travel disk is that it offers poor data consistency semantics. In some cases, the on-disk

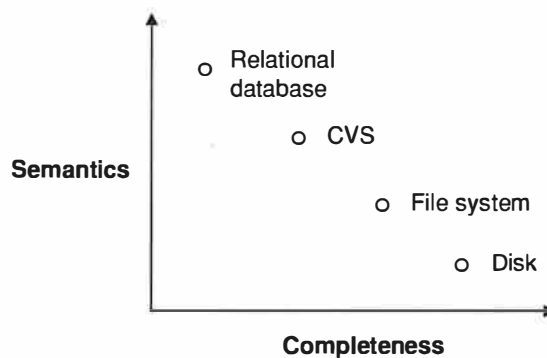


Figure 3: **Time travel storage layer tradeoff:** Chronus uses a time travel disk, which achieves completeness while forfeiting high-level semantics.

state may be corrupt, causing Chronus to discover a spurious error unrelated to the true cause of failure. More commonly, Chronus may discover the correct error, but the granularity of a block change is too fine to make a useful diagnosis. For example, configuration files can temporarily disappear while the text editor’s “save” operation is in progress. Because such inconsistencies are short-lived, it often suffices to “zoom out” by computing state changes over a slightly longer interval.

2.2 Instantiating a historical state

Another key design decision is the technique used to instantiate previous system configurations. A simple strategy would be to use application-layer restarts, in which the user-mode processes of interest are restarted after each configuration change. Unfortunately, many relevant configuration changes require whole-system reboots, including changes to system software (the kernel, shared libraries) or configuration options (TCP/IP parameters, firewall policy).

In this work, we use a virtual machine monitor (VMM) [13, 33, 35] to perform “virtual reboots” in software. Because VMMs emulate the hardware layer, they provide a more complete representation of whole-system behavior. As well, VM restarts offer a series of advantages compared to physical machine restarts. VMs can be rebooted faster, because they avoid re-initializing physical I/O devices. For Chronus, this translates into faster problem diagnosis. VMMs provide robust mechanisms for terminating failed tests and reclaiming state changes, and they enable debugger-like functionality, allowing the user to inspect or modify VM state.

There are disadvantages to using VMMs. Virtualization imposes performance overhead; this can be minimized [4], but may still be significant in some settings. VMMs tend to reduce virtual device interfaces to the lowest common denominator, and thus may mask or per-

turb some configuration errors. A VMM might not expose a bleeding-edge graphics card, for example. Finally, a VMM-based implementation of Chronus cannot diagnose configuration errors within the virtualization layer itself, such as updates to physical device drivers.

2.3 Testing a historical state

Chronus's automated diagnosis capability relies on a user-supplied software *probe* to test whether the system is functioning correctly. Testing a system is often easier than performing a full failure diagnosis. Nevertheless, testing itself can be a non-trivial task, and probe authorship represents a hurdle to utilizing Chronus.

In our current prototype, probes are written on the fly in response to specific failure conditions. We assume that troubleshooters have knowledge of shell scripts and basic command line tools. With this, many configuration errors are testable, including application crashes, a Web browser that fails to load pages, or a remote execution service that refuses access to valid clients.

For errors that are beyond the scope of shell scripts, Chronus supports a *manual testing* mode, in which the human troubleshooter performs some or all of the testing process by hand. We have found manual testing particularly useful to evaluate errors that involve sequences of GUI actions or that require the user to interpret a visual image. Manual testing can be used with more configuration errors than probes, but it imposes a heavier burden.

In the future, we plan to explore techniques to simplify probe creation. One option is to create static libraries of probes, which could be used to test generic forms of application behavior. For example, a generic web server probe might attempt to download the system home page. For graphical applications, Chronus could leverage point-and-click tools for capturing and replaying sequences of GUI actions [20].

Regardless of testing strategy, there are some configuration errors that Chronus cannot diagnose. Non-deterministic errors (or Heisenbugs [17]) that cannot be reliably reproduced are beyond the scope of our tool.

2.4 Searching over time

Given a probe, a naïve approach to finding a fault point is to sequentially examine every historical state of the system. Of course, this is impractical, as it would require instantiating, booting, and testing a virtual machine for each disk block write that occurred in the past.

A more intelligent approach is to use a binary search through time. If the troubleshooter can identify a past instance in time at which the system worked, and assuming there is a single transition from that working state to the current non-working state (as in Figure 1), then binary search will find the fault point in logarithmic time.

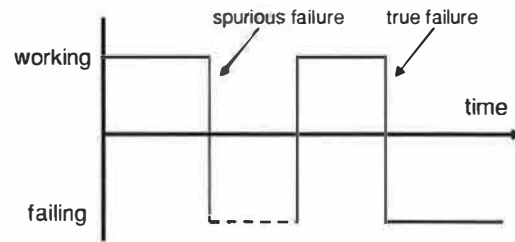


Figure 4: A **spurious search result**: Chronus may detect an error that is unrelated to the current cause of failure.

However, in some cases, a system may make *multiple* transitions from a working to a non-working state, as shown in Figure 4. Most of these additional state transitions are *spurious*, in that they are not related to the true source of the current configuration error. For example, because software is typically unavailable during a software upgrade, Chronus may mistakenly implicate a past upgrade that is unrelated to the current configuration error. Other sources of spurious errors include configuration changes that have already been fixed, and short-term inconsistencies due to corrupt file system state.

A simple strategy for dealing with multiple failures is simply to run Chronus multiple times. By choosing different time ranges for each search, Chronus can be made to explore different regions of the system timeline. This is philosophically similar to simulated annealing search, which uses random choices to escape local minimums [29]. The troubleshooter can then analyze all returned state transitions to determine which one is the likely source of failure.

An alternate strategy is to construct probes that are less likely to exhibit spurious errors. One useful strategy is to construct *error-directed* probes, which search for changes in the system's observable symptoms, regardless of whether the behavior is "correct." The key insight is that different failure causes often produce different failure modes. For example, one error might cause an application to hang, whereas another produces an identifiable error messages. Therefore, probes that search for a particular symptom are less likely to reveal spurious errors unrelated to the true cause of failure. We explore such a complex error scenario in Section 4.3.

2.5 Going from "when" to "why"

The output from Chronus is the instant in time the transition to a failing state occurred. Using this, the troubleshooter can determine the state change that induced the failure. In many cases, this information alone is sufficient to diagnose the configuration error.

In other cases, however, the individual state change revealed by Chronus may be insufficient to diagnose the error. For binary configuration data, there is no univer-

sal differencing mechanism that reveals the “meaning” of a state change. Another limitation is that Chronus cannot uncover the broader context in which a state change was carried out. For example, Chronus cannot associate a modification to a dynamic library with the act of installing a particular application. In these cases, reversing the single state change revealed by Chronus may be insufficient to remedy the problem.

The solution to this “semantic gap” [11] between hardware-level events and higher-level semantics lies in combining Chronus with other debugging tools. The UNIX `diff`, which reveals changes to ASCII files, is one such tool, but others may be more appropriate in certain contexts. For example, the Windows `regdiff` tool reveals changes between two snapshots of the Windows Registry. The Backtracker tool [22] performs root-cause analysis by mapping from a low-level state event to high-level user action. Another approach is to leverage existing system logs. Currently, the sheer volume of this logging makes it difficult to use, but the timing information provided by Chronus can be used to quickly zoom-in on a small cross-section of system log entries.

2.6 Summary

The Chronus tool maps from a user-provided software probe to the instant the system transitioned to a failing state. This information, in conjunction with higher-level analysis tools like `diff`, allows a troubleshooter to diagnose the cause of failure.

The design of Chronus was guided by a few basic goals. Unlike programming language debuggers, Chronus strives for low overhead during normal operation. To achieve this, our snapshot mechanism only captures storage updates rather than complete memory checkpoints. Chronus also strives to capture the most possible configuration errors. We achieve this by using a time-travel disk (which captures *all* persistent state changes) and virtual machine monitors (which reproduce the entire system boot sequence). Finally, Chronus strives for fast problem diagnosis. Binary search provides for diagnosis time that scales logarithmically with log size. Also, our use of virtual machines enables individual tests to execute significantly faster than would be possible on physical hardware.

3 Implementation

In this section, we describe our prototype implementation of Chronus. Our prototype consists of roughly 2600 commented lines of C code, approximately half of which is dedicated to the time-travel disk. The other half comprises the search, testing, and diagnosis functionality. Figure 5 shows a high-level view of Chronus.

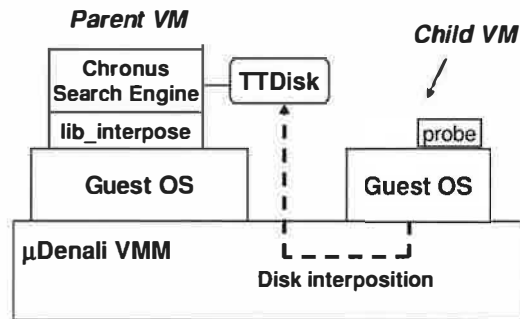


Figure 5: **Chronus software architecture:** During normal operation, the parent VM records the child’s disk writes to a time-travel disk (TTDisk). During debugging, a software probe is used to determine the correctness of a given state. Chronus uses the probe to implement a search strategy (such as binary search) across the system time-line.

Chronus makes heavy use of the μ Denali VMM [35]. Presently, μ Denali (and hence Chronus) only supports the NetBSD guest OS. μ Denali VMM allows a “parent” virtual machine to exert control over its “child” virtual machines. In addition to being able to create, destroy, and boot child VMs, the parent can interpose on and respond to its children’s virtual hardware device events. For example, if a child issues a virtual disk write, that event is passed to the parent via the “lib_interpose” interposition library. In Chronus, the child executes normal user programs, while the parent implements the Chronus debugging functionality. Chronus itself runs as a normal user process with permission to access the interposition and control APIs described by Whitaker et al. [35]

Chronus exposes a command-line interface to the troubleshooter. The `search` command initiates a diagnosis session. The command’s arguments include the name of a time-travel disk, the beginning and end of a search range (expressed as log indexes), and a probe configuration file, which defines the executable probe routine and other probe meta-data. If the search range limits are omitted, Chronus defaults to the beginning and end of the log. After Chronus has identified the instant of failure, the `attach` command is used to mount the child disk into the parent’s local file system before and after the failure. The troubleshooter can then use commands such as `diff` to extract meaningful state changes. In addition to the `search` and `attach` commands, Chronus provides a set of command line utilities for interacting with time-travel disks. See Table 1 for details.

Beyond the μ Denali VMM, the major components of Chronus’s implementation are a time-travel disk for recreating previous states, testing infrastructure for evaluating individual states, and binary search for efficiently localizing the failure across many previous states.

Category	Command	Description
Automatic search	search	binary search over a time range
	scan	linear search over a time range
	test	test a single time step
Manual search	load	load a TTDisk at one or more time steps
	attach	mount loaded disk(s) into the local file system
	boot	boot a virtual machine from a loaded disk
	kill	kill a virtual machine
Administration	make	create a new TTDisk
	query	query meta-data about a TTDisk
	flush	flush and reclaim a portion of the log

Table 1: **Chronus command-line utilities:** Automatic commands perform time-travel searches given a search probe. Manual commands allow the troubleshooter to instantiate a time-travel disk at some point in the past. Administrative commands perform TTDisk creation and maintenance.

3.1 Time-travel Disk

The Chronus time-travel disk (or TTDisk) maintains a log of the child VM’s disk writes. The TTDisk implements the μ Denali disk interface [35], a C API that allows the programmer to implement custom functionality for disk reads and writes. The TTDisk functionality is hidden behind the hardware disk interface, so the child’s guest OS requires no modifications.

The TTDisk uses two helper disks to maintain state. A *checkpoint disk* contains the initial disk contents. All disk writes are recorded to a *log disk*. The implementation of both disks is abstracted away behind the μ Denali disk interface. In our current implementation, checkpoint/log disks can be backed by either physical disk partitions or by files in the parent’s local file system. We disable write caching to ensure that disk writes are synchronously flushed to disk. Periodically, the log can be trimmed by flushing old entries back to the checkpoint.

In addition to the data disks, the TTDisk requires a *meta-data* region to map a given disk block to a location in either the checkpoint or the log. The TTDisk meta-data is similar to the checkpoint region of the log-structured file system [28], except that it preserves *all* previous disk writes, not merely those that are still active. For each TTDisk block, the meta-data region maintains a sorted list of the log writes that modified the given block. The meta-data region is backed by a file in the parent’s local file system. As with the log-structured file system, we alternate between two meta-data regions (files) to ensure consistency in the face of failure [28].

3.1.1 Design details

The TTDisk uses a block size larger than the disk sector size to reduce the amount of meta-data. For

NetBSD, the correct choice for this parameter is not the file system block size, but rather the file system fragment size. BSD systems typically use a large block size and rely on smaller fragments to efficiently store small files [23]. By choosing the TTDisk block size to match the file system fragment size, we avoid degrading performance for small writes.

A general problem for log-structured storage systems is maintaining consistency without synchronously writing log meta-data. The design of the TTDisk avoids synchronous meta-data writes by appending a *recovery sector* to each block written to the log. The recovery sector contains two fields: the virtual block index that the log write corresponds to, and a 64-bit counter, which is used to indicate the last log entry. During recovery, we roll forward the log starting from the last meta-data checkpoint until we reach a recovery sector that does not contain a valid counter.

The implementation of TTDisk crash recovery is not complete in our current prototype. We have implemented a version of TTDisk that writes recovery sectors, but this version exhibits poor performance because the μ Denali disk interface currently supports only 4 KB block operations (as opposed to 512 byte sector operations).

3.2 Testing infrastructure

Chronus relies on user-supplied software probes to indicate whether a given time step corresponds to a “correct” system state. Given such a probe, the testing infrastructure automates the task of instantiating and evaluating a previous system state. After the test has completed, any state changes made during the test are discarded.

Chronus supports two styles of software probes. *Internal* probes run inside the child virtual machine being tested. *External* probes run on the parent virtual machine conducting the test. Generally, external probes are used for diagnosing server failures. Running a probe internally on the server could yield incorrect results, since the local loopback network device is configured separately from the external interface. Internal probes are used for all other types of applications, including network clients and non-networked applications.

The steps for executing a probe differ for internal versus external probes. In both cases, the first step is to wrap the time-travel disk with a copy-on-write (COW) disk. This provides a convenient mechanism for discarding state changes made during probe execution. For internal probes, the parent virtual machine then executes a *pre-processing* routine, which mounts the COW disk into the parent’s file system, and configures the child’s file system to execute the probe routine on boot. By convention, the probe output is stored in a particular file for later extraction. Once the probe has executed, the child VM performs a halt operation, causing the parent VM to

terminate it. Alternately, a timeout mechanism is used for tests that hang or stall. After termination, the parent VM once again mounts the COW disk, and executes a *post-processing* routine to extract the probe result.

The steps for executing external probes are similar, but simpler. The pre-processing and post-processing phases are omitted. The probe runs in the parent virtual machine *while* the child virtual machine is running. Once the probe terminates or times out, the child VM is garbage collected.

3.3 Binary search

Chronus uses binary search to quickly find the fault point along the system time-line. We assume the system exhibits a transition from a working to a non-working state, as shown in Figure 1. Chronus begins by running the probe at the limits of the user-provided search range. Assuming the limits exhibit different probe results, Chronus then tests the midpoint; if the midpoint's output is the same as the endpoint's, Chronus recursively tests the earlier half of time line. If the probe's output differs from the endpoint's, Chronus recursively tests the later half of the time line. In some cases, a probe may fail to execute or may produce non-binary results. To handle this, Chronus considers all results that differ from the endpoint to be the same. This tends to work because probe failures often coincide with a non-working system, and we are generally interested in the last transition from a working to a non-working state.

Chronus requires the troubleshooter to specify a search range whose limits exhibit different probe results. Because the troubleshooter might not know an appropriate range a priori, Chronus provides a `test` command, which allows the troubleshooter to guess-and-check individual time steps. In our experience, this mechanism has proven sufficient to quickly discover a valid search range for most failure cases.

4 Debugging Experience

In this section, we describe our experience using the Chronus tool. For each experiment, we used binary search to locate the failure in time and the UNIX `diff` utility to extract the state change. In some cases, it was necessary to compute the state difference over a time range larger than a single block. As a result, `diff` sometimes detects spurious changes such as changes to emacs backup files or modifications to the system lost+found directory. In some cases, we have sanitized the results for brevity, but we never removed more than eight lines of output. All probes are written as UNIX shell scripts.

```
#!/bin/sh

TEMPFILE=./QXB50.tmp
rm -f ${TEMPFILE}

ssh root@10.19.13.17 'date' > ${TEMPFILE}

if (test -s ${TEMPFILE})
    then echo "SSHD UP"
else echo "SSHD DOWN"
fi

exit 0
```

Figure 6: **sshd probe:** This is the complete version of a shell script that diagnosed a configuration fault in the ssh daemon.

```
>>> search netbsd andrew.time
0000: SSHD UP    5267: SSHD DOWN    2633: SSHD UP
3950: SSHD UP    4608: SSHD UP        4937: SSHD DOWN
4772: SSHD UP    4854: SSHD UP        4895: SSHD UP
4916: SSHD UP    4926: SSHD DOWN    4921: SSHD DOWN
4918: SSHD UP    4919: SSHD UP        4920: SSHD DOWN

# attach ttdisk before and after fault
>>> attach andrew.time 4919 4920

# use recursive diff to find what changed
>>> diff -r /child1 /child2
Binary file /etc/ssh/ssh_host_key differs
```

Figure 7: **Diagnosing the sshd failure:** This terminal log shows Chronus's output for a binary search using the sshd probe. We have added comments to the raw output, preceded by '#'. After pinpointing the failure instant, we attach the time-travel disk before and after the fault, and use recursive diff to elicit the failure cause.

4.1 Randomly injected failures

We wrote a fault-injection tool called *etc-smasher* that creates typos in key system configuration files. Such errors can be difficult to diagnose because they often do not take effect until after the machine is rebooted. Once per second, *etc-smasher* chooses a random file from the `/etc` directory (which contains system and application configuration files). 90% of the time, *etc-smasher* writes back the file without modifying it; this creates “background noise” in the system. For the remaining 10%, the program changes the file in a small way, by either removing, adding, or modifying a character. To generate a sample run, we ran the program for several minutes, and observed the most obvious failure symptom.

The first two runs of this program induced the following configuration errors:

Configuration Fault #1: sshd failure. The child VM's sshd daemon does not respond to remote login requests.

Configuration Fault #2: boot failure. The child VM does not boot correctly. Instead of a login prompt, the user is asked to enter a shell name.

```

# Probe
#!/bin/sh

rm -f /TTOUTPUT
echo 'SUCCESS' > /TTOUTPUT

# Console output

% search netbsd andrew2.time

0000: SUCCESS 1607: FAILURE 0803: SUCCESS
1205: SUCCESS 1406: SUCCESS 1506: FAILURE
1456: FAILURE 1431: FAILURE 1418: FAILURE
1412: FAILURE 1409: FAILURE 1407: SUCCESS
1408: FAILURE

% attach andrew2.time 1407 1408
% diff -r --exclude '*dev*' /child1 /child2

file: /child1/etc/rc.d/bootconf.sh differs
<   conf=${_DUMMY}
>   conf=${DUMMY}

```

Figure 8: Boot failure probe and console output: The probe writes a string to a file, but only if the boot process completes successfully. Using this probe, Chronus diagnosed the failure as resulting from a change to the file `bootconf.sh`.

To diagnose the `sshd` failure, we wrote a probe that attempts to login via `ssh` and execute the `UNIX date` command. This probe (shown in Figure 6) is an external probe: it runs on the parent VM. Notice that the probe only deals with the observable symptoms of `ssh`, and not with any of its potential failure causes (TCP/IP mis-configurations, authentication failure, failure of the `ssh` daemon itself, etc.) Figure 7 shows the output of running a Chronus binary search for this error. The `ssh` fault was introduced between disk block writes 4919 and 4920 within the log. The output from `diff` indicates the error resulted from a change in the `ssh.host_key` file.

To diagnose the boot failure, we crafted a probe that writes a string into a file (see Figure 8). The probe runs internally (within the child VM), but only executes *after* the boot sequence has completed. As a result, the existence of the file `/TTOUTPUT` indicates a successful trial. If the boot process hangs, Chronus eventually terminates the virtual machine, and the trial constitutes a failure. As shown in Figure 8, Chronus correctly identified the source of the error as a small typo in the file `/etc/rc.d/bootconf.sh`.

4.2 Debugging Mozilla errors

To understand Chronus’s behavior for graphical applications, we analyzed a list of frequently asked questions for the Mozilla Web browser [26]. The questions fall into two categories: 1) customization questions such as “how can I make Mozilla my default browser?” and 2) errors/problems. The latter category comprises 24 out of a total of 53 questions.

In Table 2, we indicate which Mozilla errors could be diagnosed with Chronus. To qualify for Chronus support, an error must be both easily reproducible *and* result from a state change from Mozilla’s default configuration. Overall, 15 of the 24 errors (63%) in the Mozilla FAQ satisfy these criteria.

We further break down the errors captured by Chronus according to the best available testing strategy. For 7 error cases, it would be possible to construct a shell-script *probe* to elicit the failure condition. From a script, it is possible to direct Mozilla to a specific page and extract the returned result. Also, Mozilla supports a “ping” command, which is useful for determining if the application has crashed or hung. The 8 remaining error cases require *manual* control over some or all of the testing process; typically, these errors involve GUI interactions that are difficult to script. In the future, it may be possible to automate more diagnoses using graphical capture/replay tools [20].

The “connection refused” error requires further explanation. The error arises when a local firewall prevents the Mozilla executable from establishing out-bound connections. This error has a subtle dependence on the order that the firewall and Mozilla are installed. If Mozilla is installed first, then the installation of the firewall will trigger a failure, which Chronus can detect. If the firewall is installed first, then Mozilla will never work correctly. Nevertheless, it is still possible to diagnose this error with Chronus by using a probe that first *installs* Mozilla, and then tests the application.

Beyond studying applicability, we also used Chronus to diagnose several of the Mozilla errors. For each trial, we synthetically injected the error condition based on the description in the Mozilla FAQ. We then wrote a probe to diagnose the behavior, and ran Chronus to pinpoint the offending state transition. We now describe two such trials in more depth.

4.2.1 JavaScript error

JavaScript is used by some web sites to provide enhanced functionality beyond static content. JavaScript is also a security concern, and Mozilla allows users to limit the functionality of scripts, or to disable JavaScript completely. In some cases, JavaScript-enabled sites may demonstrate strange behavior if JavaScript is not enabled. For example, the user may be unable to follow hyperlinks for a particular page [26].

To model this error, we installed Mozilla in a virtual machine and disabled JavaScript through the preferences menu. To test for the error, we wrote a probe that directs Mozilla to fetch a web page that requires JavaScript support. The probe asks the user whether the resulting display output is correct. The probe and console output are shown in Figure 9.

Symptom	Cause	Chronus Support?	Testing Strategy	Comment
File space exhausted error	Bug	no		Broken by default
Can't save password	Server policy	no		Remote policy
Periodic crashes	JVM bugs	no		Not easily reproducible
Links do not work	Disabled javascript	yes	manual	Must test GUI output
Can't save preferences	Broken file path	yes	manual	Requires GUI action
Connection refused	Firewall	yes	probe	Requires install probe
Connection refused	Proxy settings	no		Broken by default
Application does not start	DLL collision	yes	probe	
Can't install extensions	Installations are disabled	yes	manual	Requires GUI action
Periodically jarbled display	Bug	no		Not easily reproducible
Installation failure	Broken install script	no		Broken by default
Scroll wheel doesn't work	Bug	no		Broken by default
Deterministic crash	Version clash	yes	probe	
User prompted for profile	Profile file is locked	yes	probe	
Copious error messages	Corrupted config file	yes	probe	
Saved files have .mp3 extension	Bad MIME type config	yes	manual	Requires GUI action
Random freezes	Corrupted config file	yes	manual	Not easily reproducible
Menu options unavailable	Bug	no		Broken by default
Can't open local files	Bug	no		Broken by default
Redirection limit exceeded	Cookies disabled	yes	probe	
Lost profile information	OS upgrade	yes	manual	Must test GUI output
Back/forward buttons grayed out	History size set to zero	yes	manual	Must test GUI output
Image links do not load	HTTP pipelining enabled	yes	manual	Must test GUI output
Home page not displayed	Adware	yes	probe	

Table 2: **The applicability of Chronus for Mozilla errors:** 15 of these 24 errors could be captured by Chronus. This means that they are both repeatable and result from a state change. In 7 of these cases, the testing could be conducted automatically given a shell-script probe. For the other 8 cases, the testing process requires assistance from a human operator, either to manipulate Mozilla or interpret its visual output.

4.2.2 A misbehaved extension

Mozilla allows developers to provide new functionality via an extensibility API. These extensions are not well-isolated, and a misbehaved extension can cause the overall browser to malfunction. To model this error, we installed a set of extensions from the Web. After quitting and restarting the program, we discovered that one of these extensions had introduced a malfunction, such that Mozilla would hang before displaying a page.

To diagnose this error, we wrote a probe that uses the Mozilla “ping” command to indicate whether a previously-launched browser is functioning correctly. Figure 10 shows the output of the `diff` utility. Although more verbose than previous examples, the state change reveals that the “StockTicker” extension caused Mozilla to malfunction.

4.3 A complex Apache error

As discussed in Section 2.4, binary search can fail in the presence of multiple faults in a single time-line. To explore this phenomenon, we introduced a sequence of configuration events inside an Apache web server, as shown in Figure 11a. The “true failure” is a mis-configuration of the Apache `suexec` command, which allows an administrator to run CGI scripts as a different

user than the overall Web server. `suexec` is a common source of configuration errors, especially when scripts require special privileges [7]. In our example, the CGI script must connect to a back end database, which only permits access from the user `www`. As a result, Web requests for this script return an HTTP error message.

In addition to the `suexec` error, we performed two actions that affect the Web server’s functionality. Near the start of the trace, we changed the server’s IP address. Because DNS mappings are not captured in our time-travel layer, any attempt to connect to the server before the IP address change will not succeed. Subsequently, we upgraded the version of the Apache running on the server. This new build was necessary to support the `suexec` command. During the installation of the upgrade, the Web server is unavailable to Chronus probes.

There are two strategies one could take in analyzing this failure. First, one could write a *success-directed* probe, which tests whether the system successfully handles requests. We wrote such a probe by testing for a successful HTTP response. The drawback of such a probe is that it may detect *any* transition from a working to a failing state, as shown in Figure 11a. In the worst case, the user must decipher two spurious results before revealing the true source of the error.

An alternate approach is to construct a *failure-*

```
# Probe
#!/bin/sh

ssh -X root@10.19.13.79 'mozilla $WEBSITE' &

echo -n 'RESULT: '

read result
echo $result

# Console output

169904: RESULT: GOOD 222044: RESULT: BAD
195974: RESULT: BAD 182939: RESULT: BAD
176421: RESULT: BAD 173162: RESULT: GOOD
174791: RESULT: BAD 173976: RESULT: BAD
173569: RESULT: BAD 173365: RESULT: GOOD
173467: RESULT: BAD 173416: RESULT: GOOD
173441: RESULT: GOOD 173454: RESULT: BAD
173447: RESULT: GOOD 173450: RESULT: GOOD
173452: RESULT: BAD 173451: RESULT: BAD

>>> diff -r /child1 /child2
file /root/.mozilla/default/zclu3kp2.slt/prefs.js
differs:
> user_pref("browser.download.dir", "/root");
> user_pref("browser.startup.homepage",
"http://www.mozilla.org/start/");
> user_pref("javascript.enabled", false);
```

Figure 9: Mozilla JavaScript probe and console output:

This probe, combined with user input, diagnosed a Mozilla rendering problem related to JavaScript. The probe runs externally on the parent VM, so that X-windows ssh forwarding is set up properly. Spurious information exists because Mozilla atomically saves all preference changes made during a user session.

directed probe. Instead of looking for successful completion of a request, a failure-directed probe searches for the precise error behavior exhibited by the application. In this example, the `suexec` failure returned a distinctive error message. Because different errors often exhibit different symptoms, a failure directed probe can result in fewer state transitions over an equivalent system timeline (see Figure 11b). Using a failure-directed probe, we discovered the source of the `suexec` failure using a single Chronus search invocation.

4.4 Reverse debugging

Although we intended Chronus as a tool for finding configuration bugs, an alternate use is to search for configuration *fixes*. This is especially useful in cases when the “fix” was applied serendipitously. For example, application X might install a dynamic library that fortuitously allows application Y to work correctly. In practice, the issue is even more subtle, because the order in which packages are installed can affect the system’s final configuration [19]. Given a failing machine and a correct machine, an administrator can use Chronus to find the fix from the correct machine, and then apply the fix to the failing machine.

```
>>> diff -r /child1 /child2
file /root/.mozilla/default/zclirw5u.slt/chrome
/chrome.rdf differs:

> <RDF:Description about="urn:mozilla:package
:stockticker"
> c:baseUrl="jar:file:///root/.mozilla/default
/zclirw5u.slt
> /chrome/stockticker.jar!/content/"
> c:locType="profile"
> c:author="Jeremy Gillick"
> c:authorURL="http://jgillick.nettripper.com/"
> c:description="Shows your favorite stocks in a
> customized ticker."
> c:displayName="StockTicker 0.4.2"
> c:extension="true"
> c:name="stockticker"
> c:settingsURL="chrome://stockticker/content
/options.xul" />
```

Figure 10: Console output for a buggy Mozilla extension: Chronus traced the failure to the “StockTicker” extension.

We used reverse debugging to elicit the correct configuration for the NetBSD Network Time Protocol (NTP) daemon. Initially, the system’s NTP configuration was incorrect, causing the system’s time to be set to an incorrect value. Although we fixed the problem in one particular VM, the change was not propagated back to the base disk image. To locate the fix, we wrote a probe that searches for unusual behavior from the `make` utility; `make` relies on a correct clock, and may force unnecessary recompilation when the clock is mis-configured.

5 Quantitative Evaluation

In this section, we provide quantitative measurements of Chronus. We analyze time-travel disk performance, log growth, and debugging execution time. All tests were run on a uniprocessor 3.2GHz Pentium 4 with hyperthreading disabled. The test machine had 2 GB of RAM, but the virtual machines (both the parent and the child) were configured to use at most 512 MB. The machine contained a single 80 GB, 7200 RPM Maxtor DiamondMax Plus IDE drive, and an Intel PRO/1000 PCI gigabit Ethernet card.

All of the following experiments were run without appending recovery sectors to log writes. Therefore, the results model a system that uses some other mechanism for insuring meta-data consistency (e.g., non-volatile RAM). An implementation with recovery sectors would require 12.5% more disk space (one 512 byte recovery sector is appended to each 4 KB block). The performance overhead would likely be similar.

5.1 Runtime Overhead

To evaluate time-travel disk performance, we ran the set of workloads shown in Table 3. We generated the sequential read and write workloads using the UNIX `dd`

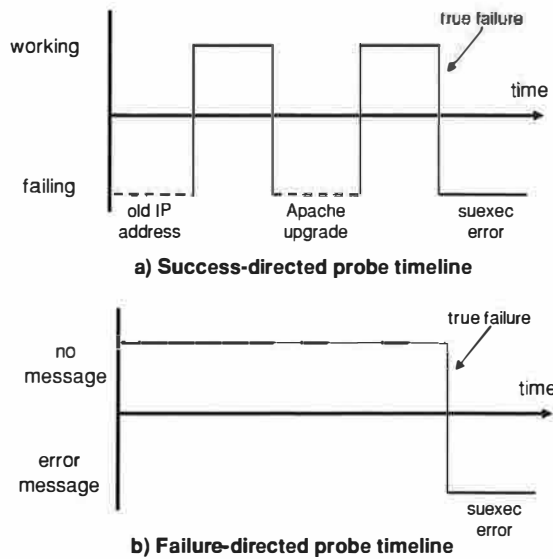


Figure 11: Apache suEXEC error, as seen by two different probes: A success-directed probe searches for transitions from a working to a failing state. This may return spurious results when the system contains multiple such transitions. A failure-directed probe searches for changes in the specific error symptom exhibited by the application. For this example, a failure-directed probe revealed the configuration error with a single Chronus invocation.

command using 32 KB block increments. We also ran an “adversarial” sequential read, in which we read over a disk region that was previously written in reverse order in 32 KB increments. Finally, we ran `untar` and `grep` over the Mozilla 1.6 source tree. Mozilla 1.6 contains 35,186 files in 2,454 directories, and has a total size of 300 MB. The “native disk” data series shows the performance of a child VM using a physical disk partition. The time-travel disk log was backed by a physical partition. No disk operations were processed by the checkpoint disk, and swapping was disabled for these tests.

For most workloads, the performance of the time-travel disk is competitive with the native disk. The one exception is the adversarial sequential read workload. Because blocks are written out in reverse log order, this style of workload generates poor performance from a log-structured storage layer. Most files are processed sequentially [2], suggesting this style of workload occurs rarely in practice.

5.2 Measuring log inflation

Chronus relies on excess storage capacity to maintain the time-travel log. This is reasonable, given that storage capacity is growing at an annual rate of 60% [18] and shows no signs of abating. Other researchers have noted that users can already go years without reclaiming storage [16].

Workload	Native disk	Time-travel disk
Sequential write	32.6 MB/sec	31.7 MB/sec
Sequential read	33.1 MB/sec	32.7 MB/sec
Sequential read (adversarial)	33.1 MB/sec	15.3 MB/sec
Untar	123.4 sec	125.7 sec
Grep	221 sec	253 sec

Table 3: Time-travel disk performance: The time-travel disk is competitive with the native disk for all workloads, except for the “adversarial” workload designed to exhibit poor locality in the time-travel log.

Operation	File System Growth	Log growth	Log growth (compressed)
copy mozilla.tar	214.8 MB	215.0 MB	29.6 MB
untar mozilla.tar	300.4 MB	1905 MB	36.1 MB
remove mozilla/	(-300.4 MB)	1432 MB	5.71 MB

Table 4: Log inflation: Operations that greatly modify the file system directory structure generate a large number of log writes. Fortunately, the writes are highly redundant and amenable to compression.

One remaining concern is *log inflation*, which arises from file system meta-data operations. Applications that heavily modify the directory structure can generate excessive log growth. Table 4 shows the amount of log growth required for various operations on the Mozilla 1.6 archive. As expected, simply copying the tar file does not generate undue log inflation. However, untaring Mozilla causes log growth that is more than six times larger than the growth in the underlying file system. Even worse, deleting the Mozilla directory tree (with `rm -Rf`) generates 1432 MB of log data! The source of this log growth is repeated, synchronous updates to file system structures such as free block lists, inodes, and directory contents.

We have considered two possibilities for combating log inflation. One possibility is compression. The contents of meta-data operations are highly redundant, and therefore would exhibit significant size reductions (as shown in Figure 4). A second possibility is to temporarily deactivate versioning — for example, using heuristics similar to those employed by the Elephant file system [30]. We have not yet experimented with or implemented either of these strategies.

5.3 Debug execution time

Because Chronus uses binary search, it can discover configuration errors in a logarithmic number of steps. Figure 12 shows Chronus’s convergence time for logs of various sizes. The test uses an internal probe that tests

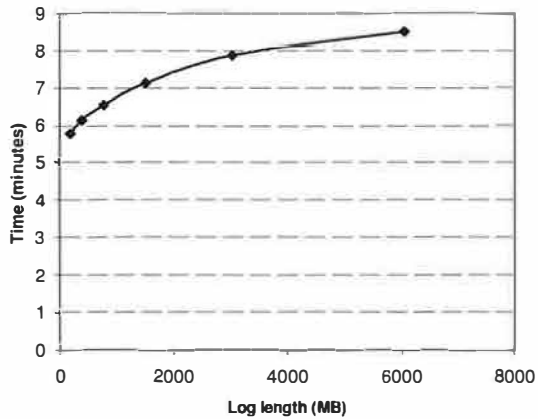


Figure 12: **Debug execution time:** The runtime grows logarithmically with log size.

for the existence of a particular file. Chronus currently requires roughly 20 seconds to conduct a single probe. More than half this time is devoted to file system consistency check (fsck) operations, which we must do twice for each probe — once before installing the probe, and once to extract the result. Moving to a journaling file system would substantially reduce this overhead.

6 Related Work

We now discuss related work in problem diagnosis and resolution. We first discuss history-based resolution techniques, and then we discuss other techniques.

6.1 History-based problem resolution

Researchers have proposed versioning storage systems at various levels of abstraction [25, 30, 31, 32, 14]. In some cases, recovery from configuration errors has been cited as a driving application. The VMWare virtual machine monitor [33] also supports checkpointing to enable safe recovery. Unlike Chronus, these systems do not perform failure diagnosis. As a result, the user is forced to undo *all* state changes that occurred after the error. Chronus helps to reveal the specific failure cause, enabling recovery with minimal lost state.

The Operator Undo work [8] attempts to recover lost state by invoking an application-specific replay procedure. In a similar vein, Windows XP restore [24] allows developers to exert some control over which state is included in state snapshots. Both of these approaches, being application specific, are less general than Chronus. Also, these techniques have side effects, which can further corrupt system state. For example, Windows Restore may inadvertently re-introduce a virus into the system.

The Backtracker tool [22] maintains an operating system causal history log. Such a tool could address one of Chronus's current shortcomings, which is its inability

to extract semantically relevant debugging information from the child virtual machine. For example, Chronus might discover that an application failure was caused by an update to a particular dynamic library. Given this starting point, a Backtracker-like tool could determine that the library change was caused by the installation of an unrelated application.

Several research efforts have extended programming language debuggers with the ability to perform time-travel or backwards execution [6]. These systems tend to have high overhead or long replay times, depending on the extent to which they rely on checkpointing or logging. In addition, these systems are tied to a particular language or runtime environment. Chronus detects configuration errors that span applications and the OS, and it does so with tolerable overhead by recording only those changes that reach stable storage.

Delta-debugging [36] applies search techniques to the problem of localizing source code edits that induced a failure. Delta-debugging does not assume changes are ordered, and much of the system's complexity derives from having to prune an exponentially large search space. The challenges for Chronus relate to capturing and replaying complete system states using time-travel disks and virtual machines.

The STRIDER [34] project uses periodic snapshots of the Windows registry to reveal configuration errors. Unlike Chronus, STRIDER monitors a *single* execution of a failing program, during which it records the registry keys that are accessed by the faulty process. STRIDER requires registry-specific heuristics to prune the search space: for example, registry keys that differ across machines are less likely to be at fault. STRIDER does not detect indirect dependencies that result from interactions with helper processes or the operating system. For example, STRIDER cannot reveal errors related to TCP/IP parameters or firewall policy.

6.2 Other problem resolution techniques

A direct strategy for automated debugging is to construct a software agent that embodies the knowledge of a human expert [3]. The limitation of such systems is that they are only as good as their initial diagnosis heuristics. Complex systems generate unexpected errors. Chronus can capture these errors by operating beneath the layer of operating system and application semantics.

The No-Futz [21] computing initiative advocates a principled approach to maintaining configuration state. For example, the authors advocate making individual configuration parameters orthogonal to limit the effect of unintended side effects. While this is a worthwhile goal, the tight integration of today's application and system functionality suggests that debugging techniques will still be necessary when inevitable failures occur.

Redstone et al. proposed a model of automated debugging that extracts relevant system state and symptoms to serve as a query against a database of known problems [27]. A challenge for such a system is constructing a database and query format that yield meaningful results. Chronus avoids using databases by directly “querying” the system state at a previous instant in time. The results returned by our system may be more relevant because they pertain exclusively to the system under consideration.

Several recent projects have investigated path-based debugging of distributed systems [10, 1]. These systems log the interactions between components or nodes of a distributed system. By applying statistical techniques to these traces, it is possible to extract some information of interest, e.g., localizing performance problems or detecting an incipient system failure. These systems depend on the ability to extract large volumes of trace data showing the integration between distributed components. Chronus is useful in situations where these assumptions are not satisfied, e.g., desktop personal computers.

7 Future Work

Although functional, our Chronus prototype could be extended in numerous ways. One area of interest is extending μ Denali and Chronus beyond a UNIX environment. In particular, systems based on Microsoft Windows are likely to exhibit qualitatively different configuration errors. We are also interested in extending Chronus with different time-travel storage mechanisms. For example, some administrators use CVS to maintain a log of configuration changes. Chronus could use CVS check-ins to reconstruct previous system states.

Chronus is not a fully automatic tool: the troubleshooter must supply a software probe and interpret the state change that induced the failure. It may be possible to reduce this manual effort by combining Chronus with related research efforts. For example, capture/replay tools could automate probe creation [20], and Backtracker [22] could simplify end-to-end diagnosis by mapping from a low-level state change to a high-level action.

A final area for future work is to perform a more complete evaluation of Chronus. Our work to date has focused on a small number of case studies representing “common” configuration errors. Although our initial results are promising, we do not have enough data about configuration errors in the wild to make strong claims about the applicability of Chronus. An even harder challenge is to measure the “usefulness” of our tool. In the end, a complete evaluation of Chronus will likely require a user study, since simulating a human operator is intractable. Work by Brown et al. provides a starting point for such an effort [9].

8 Conclusions

Software systems often break. When they do, diagnosing the cause of failure can be difficult, especially when the application depends on a wide range of system-level and user-level functionality. Existing automated approaches based on expert systems can only handle error cases that are known in advance. Human experts can leverage intuition to solve unforeseen problems, but manual diagnosis requires significant expertise, which ultimately translates into substantial cost.

This paper has described Chronus, a tool for automating the diagnosis of configuration errors caused by a state change. Chronus represents a novel synthesis of existing techniques: versioning storage systems, virtual machine monitors, testing, and search. Chronus reduces the burden on human experts from complete diagnosis (“why is the system not working?”) to testing for correctness (“is the system working?”). Our experience to date suggests that Chronus is a valuable tool for a significant class of configuration errors.

9 Acknowledgments

We thank our shepherd, Timothy Roscoe, for his guidance, and Ed Lazowska, Neil Spring, Marianne Shaw, and Andrew Schwerin for their insightful comments. We also thank Phil Levis for giving us pointers to valuable related work. This research was supported in part by NSF Career award ANI-0132817, funding from Intel Corporation, and a gift from Nortel Networks.

References

- [1] M.K. Aguilera, J.C. Mogul, J.L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, 2003.
- [2] M.G. Baker, J.H. Hartman, H.D. Kupfer, K.W. Shirriff, and J.K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, 1991.
- [3] G. Banga. Auto-diagnosis of field problems in an appliance operating system. In *Proceedings of the USENIX Annual Technical Conference*, June 2000.
- [4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th Symposium on Operating System Principles (SOSP 2003)*, Bolton Landing, NY, October 2003.
- [5] S. Beattie, S. Arnold, C. Cowan, P. Wagle, C. Wright, and A. Shostack. Timing the application of security patches for optimal uptime. In *Proceedings of the Sixteenth USENIX LISA Conference*, November 2002.
- [6] B. Boothe. Efficient algorithms for bidirectional debugging. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2000.

- [7] R. Bowen and K. Coar. *Apache Cookbook*. O'Reilly and Associates, November 2003.
- [8] A.A. Brown and D.A. Patterson. Undo for operators: Building an undoable e-mail store. In *Proceedings of the 2003 USENIX Annual Technical Conference*, June 2003.
- [9] A.B. Brown, L. Chung, W. Kakes, C. Ling, and D.A. Patterson. Experiences with evaluation human-assisted recovery processes. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2004.
- [10] M.Y. Chen, A. Accardi, E. Kiciman, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *Proceedings of the First Symposium on Network Systems Design and Implementation*, March 2004.
- [11] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 2001.
- [12] Final Report of the CRA Conference on Grand Research Challenges in Information Systems. <http://www.cra.org/reports/gc.systems.pdf>, 2003.
- [13] R.J. Creasy. The origin of the VM/370 time-sharing system. *IBM Journal of Research and Development*, 25(5), 1981.
- [14] Version management with CVS. <https://www.cvshome.org/docs/manual/>.
- [15] G.W. Dunlap, S.T. King, S. Cinar, M. Basrai, and P.M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, December 2002.
- [16] J. Gemmell, G. Bell, R. Lueder, S. Drucker, and C. Wong. MyLifeBits: Fulfilling the Memex vision. In *ACM Multimedia*, December 2002.
- [17] Jim Gray. Why do computers stop and what can be done about it? In *Proceedings of the 5th Symposium on Reliability in Distributed Software and Database systems*, January 1986.
- [18] E. Growchowski. Emerging trends in data storage on magnetic hard disk drives. *Datatech*, 1998.
- [19] J. Hart and J. D'Amelia. An analysis of RPM validation drift. In *Proceedings of the USENIX LISA Conference*, 2002.
- [20] J.H. Hicinbothorn and W.W. Zachary. A tool for automatically generating transcripts of human-computer interaction. In *Proceedings of the Human Factors and Ergonomics Society 37th Annual Meeting*, 1993.
- [21] D.A. Holland, W. Josephson, K. Magoutis, M. Seltzer, C.A. Stein, and A. Lim. Research issues in no-futz computing. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, May 2001.
- [22] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the 19th Symposium on Operating System Principles (SOSP 2003)*, Bolton Landing, NY, October 2003.
- [23] Marshall Kirk McKusick, Bill Joy, Leffler, and Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3), 1984.
- [24] Microsoft, Inc. Windows XP system restore. <http://msdn.microsoft.com/library/default.asp?URL=/library/techart/wind%owsxpsystemrestore.htm>, April 2001.
- [25] C.B. Morrey and D. Grunwald. Peabody: The time traveling disk. In *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies*, April 2003.
- [26] Mozilla FAQ: Using mozilla. http://mozilla.gunnars.net/mozfaq_use.html.
- [27] Joshua A. Redstone, Michael M. Swift, and Brian N. Bershad. Using computers to diagnose computer problems. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, 2003.
- [28] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, 1991.
- [29] S.J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, December 2002.
- [30] D.S. Santry, M.J. Feeley, N.C. Hutchinson, A.C. Veitch, R.W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, December 1999.
- [31] C.A.N. Soules, G.R. Goodson, J.D. Strunk, and G.R. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of the 2nd USENIX conference on file and storage technologies*, March 2003.
- [32] M. Stonebreaker. The design of the POSTGRES storage system. In *Proceedings of the 13th International Conference on Very Large Databases*, September 1987.
- [33] VMware, Inc. VMware virtual machine technology. <http://www.vmware.com/>.
- [34] Y. Wang, C. Verbowski, J. Dunagan, Y. Chen, H.J. Wang, C. Yuan, and Z. Zhang. STRIDER: A black-box, state-based approach to change and configuration management and support. In *Proceedings of the USENIX LISA Conference*, October 2003.
- [35] Andrew Whitaker, Richard S. Cox, Marianne Shaw, and Steven D. Gribble. Constructing services with interposable virtual hardware. In *Proceedings of the First Symposium on Network Systems Design and Implementation*, March 2004.
- [36] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *Proceedings of the 7th European Software Engineering Conference*, September 1999.

Chain Replication for Supporting High Throughput and Availability

Robbert van Renesse
rvr@cs.cornell.edu

Fred B. Schneider
fbs@cs.cornell.edu

*FAST Search & Transfer ASA
Tromsø, Norway
and
Department of Computer Science
Cornell University
Ithaca, New York 14853*

Abstract

Chain replication is a new approach to coordinating clusters of fail-stop storage servers. The approach is intended for supporting large-scale storage services that exhibit high throughput and availability without sacrificing strong consistency guarantees. Besides outlining the chain replication protocols themselves, simulation experiments explore the performance characteristics of a prototype implementation. Throughput, availability, and several object-placement strategies (including schemes based on distributed hash table routing) are discussed.

1 Introduction

A *storage system* typically implements operations so that clients can store, retrieve, and/or change data. File systems and database systems are perhaps the best known examples. With a file system, operations (read and write) access a single file and are idempotent; with a database system, operations (transactions) may each access multiple objects and are serializable.

This paper is concerned with storage systems that sit somewhere between file systems and database systems. In particular, we are concerned with storage systems, henceforth called *storage services*, that

- store *objects* (of an unspecified nature),
- support *query* operations to return a value derived from a single object, and
- support *update* operations to atomically change the state of a single object according to some

pre-programmed, possibly non-deterministic, computation involving the prior state of that object.

A file system write is thus a special case of our storage service update which, in turn, is a special case of a database transaction.

Increasingly, we see on-line vendors (like Amazon.com), search engines (like Google's and FAST's), and a host of other information-intensive services provide value by connecting large-scale storage systems to networks. A storage service is the appropriate compromise for such applications, when a database system would be too expensive and a file system lacks rich enough semantics.

One challenge when building a large-scale storage service is maintaining high availability and high throughput despite failures and concomitant changes to the storage service's configuration, as faulty components are detected and replaced.

Consistency guarantees also can be crucial. But even when they are not, the construction of an application that fronts a storage service is often simplified given *strong consistency guarantees*, which assert that (i) operations to query and update individual objects are executed in some sequential order and (ii) the effects of update operations are necessarily reflected in results returned by subsequent query operations.

Strong consistency guarantees are often thought to be in tension with achieving high throughput and high availability. So system designers, reluctant to sacrifice system throughput or availability, regularly decline to support strong consistency guarantees. The Google File System (GFS) illustrates this thinking [11]. In fact, strong consistency guarantees

in a large-scale storage service are not incompatible with high throughput and availability. And the new *chain replication* approach to coordinating fail-stop servers, which is the subject of this paper, simultaneously supports high throughput, availability, and strong consistency.

We proceed as follows. The interface to a generic storage service is specified in §2. In §3, we explain how query and update operations are implemented using chain replication. Chain replication can be viewed as an instance of the primary/backup approach, so §4 compares them. Then, §5 summarizes experiments to analyze throughput and availability using our prototype implementation of chain replication and a simulated network. Some of these simulations compare chain replication with storage systems (like CFS [7] and PAST [19]) based on distributed hash table (DHT) routing; other simulations reveal surprising behaviors when a system employing chain replication recovers from server failures. Chain replication is compared in §6 to other work on scalable storage systems, trading consistency for availability, and replica placement. Concluding remarks appear in §7, followed by endnotes.

2 A Storage Service Interface

Clients of a storage service issue *requests* for query and update operations. While it would be possible to ensure that each request reaching the storage service is guaranteed to be performed, the end-to-end argument [20] suggests there is little point in doing so. Clients are better off if the storage service simply generates a reply for each request it receives and completes, because this allows lost requests and lost replies to be handled as well: a client re-issues a request if too much time has elapsed without receiving a reply.

- The reply for `query(objId, opts)` is derived from the value of object `objId`; options `opts` characterizes what parts of `objId` are returned. The value of `objId` remains unchanged.
- The reply for `update(objId, newVal, opts)` depends on options `opts` and, in the general case, can be a value V produced in some nondeterministic pre-programmed way involving the current value of `objId` and/or value `newVal`; V then becomes the new value of `objId`.¹

Query operations are idempotent, but update operations need not be. A client that re-issues a non-idempotent update request must therefore take precautions to ensure the update has not already been

State is:

$Hist_{objID}$: update request sequence
 $Pending_{objID}$: request set

Transitions are:

T1: Client request r arrives:

$Pending_{objID} := Pending_{objID} \cup \{r\}$

T2: Client request $r \in Pending_{objID}$ ignored:

$Pending_{objID} := Pending_{objID} - \{r\}$

T3: Client request $r \in Pending_{objID}$ processed:

$Pending_{objID} := Pending_{objID} - \{r\}$

if $r = \text{query}(objId, opts)$ then

reply according options `opts` based
on $Hist_{objID}$

else if $r = \text{update}(objId, newVal, opts)$ then

$Hist_{objID} := Hist_{objID} \cdot r$

reply according options `opts` based
on $Hist_{objID}$

Figure 1: Client's View of an Object.

performed. The client might, for example, first issue a query to determine whether the current value of the object already reflects the update.

A client request that is lost before reaching the storage service is indistinguishable to that client from one that is ignored by the storage service. This means that clients would not be exposed to a new failure mode when a storage server exhibits transient outages during which client requests are ignored. Of course, acceptable client performance likely would depend on limiting the frequency and duration of transient outages.

With chain replication, the duration of each transient outage is far shorter than the time required to remove a faulty host or to add a new host. So, client request processing proceeds with minimal disruption in the face of failure, recovery, and other reconfiguration. Most other replica-management protocols either block some operations or sacrifice consistency guarantees following failures and during reconfigurations.

We specify the functionality of our storage service by giving the client view of an object's state and of that object's state transitions in response to query and update requests. Figure 1 uses pseudo-code to give such a specification for an object `objID`.

The figure defines the state of `objID` in terms of two variables: the sequence² $Hist_{objID}$ of updates that have been performed on `objID` and a set $Pending_{objID}$ of unprocessed requests.

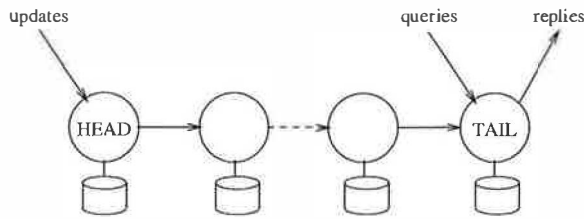


Figure 2: A chain.

Then, the figure lists possible state transitions. Transition T1 asserts that an arriving client request is added to $Pending_{objID}$. That some pending requests are ignored is specified by transition T2—this transition is presumably not taken too frequently. Transition T3 gives a high-level view of request processing: the request r is first removed from $Pending_{objID}$; query then causes a suitable reply to be produced whereas update also appends r (denoted by \cdot) to $Hist_{objID}$.³

3 Chain Replication Protocol

Servers are assumed to be fail-stop [21]:

- each server halts in response to a failure rather than making erroneous state transitions, and
- a server's halted state can be detected by the environment.

With an object replicated on t servers, as many as $t-1$ of the servers can fail without compromising the object's availability. The object's availability is thus increased to the probability that all servers hosting that object have failed; simulations in §5.4 explore this probability for typical storage systems. Henceforth, we assume that at most $t-1$ of the servers replicating an object fail concurrently.

In chain replication, the servers replicating a given object $objID$ are linearly ordered to form a *chain*. (See Figure 2.) The first server in the chain is called the *head*, the last server is called the *tail*, and request processing is implemented by the servers roughly as follows:

Reply Generation. The reply for every request is generated and sent by the tail.

Query Processing. Each query request is directed to the tail of the chain and processed there atomically using the replica of $objID$ stored at the tail.

Update Processing. Each update request is directed to the head of the chain. The request is processed there atomically using replica of $objID$ at the head, then state changes are forwarded along a reliable FIFO link to the next element of the chain (where it is handled and forwarded), and so on until the request is handled by the tail.

Strong consistency thus follows because query requests and update requests are all processed serially at a single server (the tail).

Processing a query request involves only a single server, and that means query is a relatively cheap operation. But when an update request is processed, computation done at $t-1$ of the t servers does not contribute to producing the reply and, arguably, is redundant. The redundant servers do increase the fault-tolerance, though.

Note that some redundant computation associated with the $t-1$ servers is avoided in chain replication because the new value is computed once by the head and then forwarded down the chain, so each replica has only to perform a write. This forwarding of state changes also means update can be a non-deterministic operation—the non-deterministic choice is made once, by the head.

3.1 Protocol Details

Clients do not directly read or write variables $Hist_{objID}$ and $Pending_{objID}$ of Figure 1, so we are free to implement them in any way that is convenient. When chain replication is used to implement the specification of Figure 1:

- $Hist_{objID}$ is defined to be $Hist_{objID}^T$, the value of $Hist_{objID}$ stored by tail T of the chain, and
- $Pending_{objID}$ is defined to be the set of client requests received by any server in the chain and not yet processed by the tail.

The chain replication protocols for query processing and update processing are then shown to satisfy the specification of Figure 1 by demonstrating how each state transition made by any server in the chain is equivalent either to a no-op or to allowed transitions T1, T2, or T3.

Given the descriptions above for how $Hist_{objID}$ and $Pending_{objID}$ are implemented by a chain (and assuming for the moment that failures do not occur), we observe that the only server transitions affecting $Hist_{objID}$ and $Pending_{objID}$ are: (i) a server in the chain receiving a request from a client (which affects $Pending_{objID}$), and (ii) the tail processing a client

request (which affects $Hist_{objID}$). Since other server transitions are equivalent to no-ops, it suffices to show that transitions (i) and (ii) are consistent with T1 through T3.

Client Request Arrives at Chain. Clients send requests to either the head (update) or the tail (query). Receipt of a request r by either adds r to the set of requests received by a server but not yet processed by the tail. Thus, receipt of r by either adds r to $Pending_{objID}$ (as defined above for a chain), and this is consistent with T1.

Request Processed by Tail. Execution causes the request to be removed from the set of requests received by any replica that have not yet been processed by the tail, and therefore it deletes the request from $Pending_{objID}$ (as defined above for a chain)—the first step of T3. Moreover, the processing of that request by tail T uses replica $Hist_{objID}^T$ which, as defined above, implements $Hist_{objID}$ —and this is exactly what the remaining steps of T3 specify.

Coping with Server Failures

In response to detecting the failure of a server that is part of a chain (and, by the fail-stop assumption, all such failures are detected), the chain is reconfigured to eliminate the failed server. For this purpose, we employ a service, called the *master*, that

- detects failures of servers,
- informs each server in the chain of its new predecessor or new successor in the new chain obtained by deleting the failed server,
- informs clients which server is the head and which is the tail of the chain.

In what follows, we assume the master is a single process that never fails. This simplifies the exposition but is not a realistic assumption; our prototype implementation of chain replication actually replicates a master process on multiple hosts, using Paxos [16] to coordinate those replicas so they behave in aggregate like a single process that does not fail.

The master distinguishes three cases: (i) failure of the head, (ii) failure of the tail, and (iii) failure of some other server in the chain. The handling of each, however, depends on the following insight about how updates are propagated in a chain.

Let the server at the head of the chain be labeled H , the next server be labeled $H + 1$, etc., through the tail, which is given label T . Define

$$Hist_{objID}^i \preceq Hist_{objID}^j$$

to hold if sequence⁴ of requests $Hist_{objID}^i$ at the server with label i is a prefix of sequence $Hist_{objID}^j$ at the server with label j . Because updates are sent between elements of a chain over reliable FIFO links, the sequence of updates received by each server is a prefix of those received by its successor. So we have:

Update Propagation Invariant. For servers labeled i and j such that $i \leq j$ holds (i.e., i is a predecessor of j in the chain) then:

$$Hist_{objID}^j \preceq Hist_{objID}^i.$$

Failure of the Head. This case is handled by the master removing H from the chain and making the successor to H the new head of the chain. Such a successor must exist if our assumption holds that at most $t - 1$ servers are faulty.

Changing the chain by deleting H is a transition and, as such, must be shown to be either a no-op or consistent with T1, T2, and/or T3 of Figure 1. This is easily done. Altering the set of servers in the chain could change the contents of $Pending_{objID}$ —recall, $Pending_{objID}$ is defined as the set of requests received by any server in the chain and not yet processed by the tail, so deleting server H from the chain has the effect of removing from $Pending_{objID}$ those requests received by H but not yet forwarded to a successor. Removing a request from $Pending_{objID}$ is consistent with transition T2, so deleting H from the chain is consistent with the specification in Figure 1.

Failure of the Tail. This case is handled by removing tail T from the chain and making predecessor T^- of T the new tail of the chain. As before, such a predecessor must exist given our assumption that at most $t - 1$ server replicas are faulty.

This change to the chain alters the values of both $Pending_{objID}$ and $Hist_{objID}$, but does so in a manner consistent with repeated T3 transitions: $Pending_{objID}$ decreases in size because $Hist_{objID}^T \preceq Hist_{objID}^{T^-}$ (due to the Update Propagation Invariant, since $T^- < T$ holds), so changing the tail from T to T^- potentially increases the set of requests completed by the tail which, by definition, decreases the set of requests in $Pending_{objID}$. Moreover, as required by T3, those update requests completed

by T^- but not completed by T do now appear in $Hist_{objID}$ because with T^- now the tail, $Hist_{objID}$ is defined as $Hist_{objID}^{T^-}$.

Failure of Other Servers. Failure of a server S internal to the chain is handled by deleting S from the chain. The master first informs S 's successor S^+ of the new chain configuration and then informs S 's predecessor S^- . This, however, could cause the Update Propagation Invariant to be invalidated unless some means is employed to ensure update requests that S received before failing will still be forwarded along the chain (since those update requests already do appear in $Hist_{objID}^i$ for any predecessor i of S). The obvious candidate to perform this forwarding is S^- , but some bookkeeping and coordination are now required.

Let U be a set of requests and let $<_U$ be a total ordering on requests in that set. Define a request sequence \bar{r} to be *consistent* with $(U, <_U)$ if (i) all requests in \bar{r} appear in U and (ii) requests are arranged in \bar{r} in ascending order according to $<_U$. Finally, for request sequences \bar{r} and \bar{r}' consistent with $(U, <_U)$, define $\bar{r} \oplus \bar{r}'$ to be a sequence of all requests appearing in \bar{r} or in \bar{r}' such that $\bar{r} \oplus \bar{r}'$ is consistent with $(U, <_U)$ (and therefore requests in sequence $\bar{r} \oplus \bar{r}'$ are ordered according to $<_U$).

The Update Propagation Invariant is preserved by requiring that the first thing a replica S^- connecting to a new successor S^+ does is: send to S^+ (using the FIFO link that connects them) those requests in $Hist_{objID}^{S^-}$ that might not have reached S^+ ; only after those have been sent may S^- process and forward requests that it receives subsequent to assuming its new chain position.

To this end, each server i maintains a list $Sent_i$ of update requests that i has forwarded to some successor but that might not have been processed by the tail. The rules for adding and deleting elements on this list are straightforward: Whenever server i forwards an update request r to its successor, server i also appends r to $Sent_i$. The tail sends an acknowledgement $ack(r)$ to its predecessor when it completes the processing of update request r . And upon receipt $ack(r)$, a server i deletes r from $Sent_i$ and forwards $ack(r)$ to its predecessor.

A request received by the tail must have been received by all of its predecessors in the chain, so we can conclude:

Inprocess Requests Invariant. If $i \leq j$ then

$$Hist_{objID}^i = Hist_{objID}^j \oplus Sent_i.$$

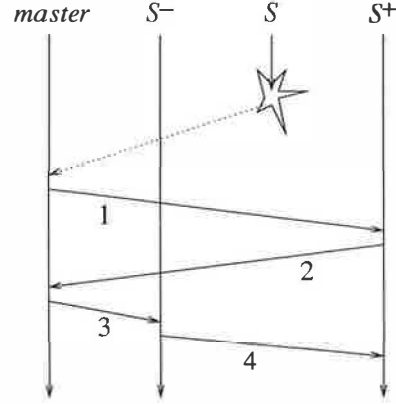


Figure 3: Space-time diagram for deletion of internal replica.

Thus, the Update Propagation Invariant will be maintained if S^- , upon receiving notification from the master that S^+ is its new successor, first forwards the sequence of requests in $Sent_{S^-}$ to S^+ . Moreover, there is no need for S^- to forward the prefix of $Sent_{S^-}$ that already appears in $Hist_{objID}^{S^+}$.

The protocol whose execution is depicted in Figure 3 embodies this approach (including the optimization of not sending more of the prefix than necessary). Message 1 informs S^+ of its new role; message 2 acknowledges and informs the master what is the sequence number sn of the last update request S^+ has received; message 3 informs S^- of its new role and of sn so S^- can compute the suffix of $Sent_{S^-}$ to send to S^+ ; and message 4 carries that suffix.

Extending a Chain. Failed servers are removed from chains. But shorter chains tolerate fewer failures, and object availability ultimately could be compromised if ever there are too many server failures. The solution is to add new servers when chains get short. Provided the rate at which servers fail is not too high and adding a new server does not take too long, then chain length can be kept close to the desired t servers (so $t - 1$ further failures are needed to compromise object availability).

A new server could, in theory, be added anywhere in a chain. In practice, adding a server T^+ to the very end of a chain seems simplest. For a tail T^+ , the value of $Sent_{T^+}$ is always the empty list, so initializing $Sent_{T^+}$ is trivial. All that remains is to initialize local object replica $Hist_{objID}^{T^+}$ in a way that satisfies the Update Propagation Invariant.

The initialization of $Hist_{objID}^{T^+}$ can be accom-

plished by having the chain's current tail T forward the object replica $Hist_{objID}^T$ it stores to T^+ . The forwarding (which may take some time if the object is large) can be concurrent with T 's processing query requests from clients and processing updates from its predecessor, provided each update is also appended to $Sent_T$. Since $Hist_{objID}^{T^+} \preceq Hist_{objID}^T$ holds throughout this forwarding, Update Propagation Invariant holds. Therefore, once

$$Hist_{objID}^T = Hist_{objID}^{T^+} \oplus Sent_T$$

holds, Inprocess Requests Invariant is established and T^+ can begin serving as the chain's tail:

- T is notified that it no longer is the tail. T is thereafter free to discard query requests it receives from clients, but a more sensible policy is for T to forward such requests to new tail T^+ .
- Requests in $Sent_T$ are sent (in sequence) to T^+ .
- The master is notified that T^+ is the new tail.
- Clients are notified that query requests should be directed to T^+ .

4 Primary/Backup Protocols

Chain replication is a form of primary/backup approach [3], which itself is an instance of the state machine approach [22] to replica management. In the primary/backup approach, one server, designated the *primary*

- imposes a sequencing on client requests (and thereby ensures strong consistency holds),
- distributes (in sequence) to other servers, known as *backups*, the client requests or resulting updates,
- awaits acknowledgements from all non-faulty backups, and
- after receiving those acknowledgements then sends a reply to the client.

If the primary fails, one of the back-ups is promoted into that role.

With chain replication, the primary's role in sequencing requests is shared by two replicas. The head sequences update requests; the tail extends that sequence by interleaving query requests. This sharing of responsibility not only partitions the sequencing task but also enables lower-latency and

lower-overhead processing for query requests, because only a single server (the tail) is involved in processing a query and that processing is never delayed by activity elsewhere in the chain. Compare that to the primary backup approach, where the primary, before responding to a query, must await acknowledgements from backups for prior updates.

In both chain replication and in the primary/backup approach, update requests must be disseminated to all servers replicating an object or else the replicas will diverge. Chain replication does this dissemination serially, resulting in higher latency than the primary/backup approach where requests were distributed to backups in parallel. With parallel dissemination, the time needed to generate a reply is proportional to the maximum latency of any non-faulty backup; with serial dissemination, it is proportional to the sum of those latencies.

Simulations reported in §5 quantify all of these performance differences, including variants of chain replication and the primary/backup approach in which query requests are sent to any server (with expectations of trading increased performance for the strong consistency guarantee).

Simulations are not necessary for understanding the differences in how server failures are handled by the two approaches, though. The central concern here is the duration of any transient outage experienced by clients when the service reconfigures in response to a server failure; a second concern is the added latency that server failures introduce.

The delay to detect a server failure is by far the dominant cost, and this cost is identical for both chain replication and the primary/backup approach. What follows, then, is an analysis of the recovery costs for each approach assuming that a server failure has been detected; message delays are presumed to be the dominant source of protocol latency.

For chain replication, there are three cases to consider: failure of the head, failure of a middle server, and failure of the tail.

- **Head Failure.** Query processing continues uninterrupted. Update processing is unavailable for 2 message delivery delays while the master broadcasts a message to the new head and its successor, and then it notifies all clients of the new head using a broadcast.
- **Middle Server Failure.** Query processing continues uninterrupted. Update processing can be delayed but update requests are not lost, hence no transient outage is experienced, provided some server in a prefix of the chain that has received the request remains operating.

Failure of a middle server can lead to a delay in processing an update request—the protocol of Figure 3 involves 4 message delivery delays.

- **Tail Failure.** Query and update processing are both unavailable for 2 message delivery delays while the master sends a message to the new tail and then notifies all clients of the new tail using a broadcast.

With the primary/backup approach, there are two cases to consider: failure of the primary and failure of a backup. Query and update requests are affected the same way for each.

- **Primary Failure.** A transient outage of 5 message delays is experienced, as follows. The master detects the failure and broadcasts a message to all backups, requesting the number of updates each has processed and telling them to suspend processing requests. Each backup replies to the master. The master then broadcasts the identity of the new primary to all backups. The new primary is the one having processed the largest number of updates, and it must then forward to the backups any updates that they are missing. Finally, the master broadcasts a message notifying all clients of the new primary.
- **Backup Failure.** Query processing continues uninterrupted provided no update requests are in progress. If an update request is in progress then a transient outage of at most 1 message delay is experienced while the master sends a message to the primary indicating that acknowledgements will not be forthcoming from the faulty backup and requests should not subsequently be sent there.

So the worst case outage for chain replication (tail failure) is never as long as the worst case outage for primary/backup (primary failure); and the best case for chain replication (middle server failure) is shorter than the best case outage for primary/backup (backup failure). Still, if duration of transient outage is the dominant consideration in designing a storage service then choosing between chain replication and the primary/backup approach requires information about the mix of request types and about the chances of various servers failing.

5 Simulation Experiments

To better understand throughput and availability for chain replication, we performed a series of ex-

periments in a simulated network. These involve prototype implementations of chain replication as well as some of the alternatives. Because we are mostly interested in delays intrinsic to the processing and communications that chain replication entails, we simulated a network with infinite bandwidth but with latencies of 1 ms per message.

5.1 Single Chain, No Failures

First, we consider the simple case when there is only one chain, no failures, and replication factor t is 2, 3, and 10. We compare throughput for four different replication management alternatives:

- **chain:** Chain replication.
- **p/b:** Primary/backup.
- **weak-chain:** Chain replication modified so query requests go to any random server.
- **weak-p/b:** Primary/backup modified so query requests go to any random server.

Note, **weak-chain** and **weak-p/b** do not implement the strong consistency guarantees that **chain** and **p/b** do.

We fix the query latency at a server to be 5 ms and fix the update latency to be 50 ms. (These numbers are based on actual values for querying or updating a web search index.) We assume each update entails some initial processing involving a disk read, and that it is cheaper to forward object-differences for storage than to repeat the update processing anew at each replica; we expect that the latency for a replica to process an object-difference message would be 20 ms (corresponding to a couple of disk accesses and a modest computation).

So, for example, if a chain comprises three servers, the total latency to perform an update is 94 ms: 1 ms for the message from the client to the head, 50 ms for an update latency at the head, 20 ms to process the object difference message at each of the two other servers, and three additional 1 ms forwarding latencies. Query latency is only 7 ms, however.

In Figure 4 we graph total throughput as a function of the percentage of requests that are updates for $t = 2$, $t = 3$ and $t = 10$. There are 25 clients, each doing a mix of requests split between queries and updates consistent with the given percentage. Each client submits one request at a time, delaying between requests only long enough to receive the response for the previous request. So the clients together can have as many as 25 concurrent requests outstanding. Throughput for **weak-chain**

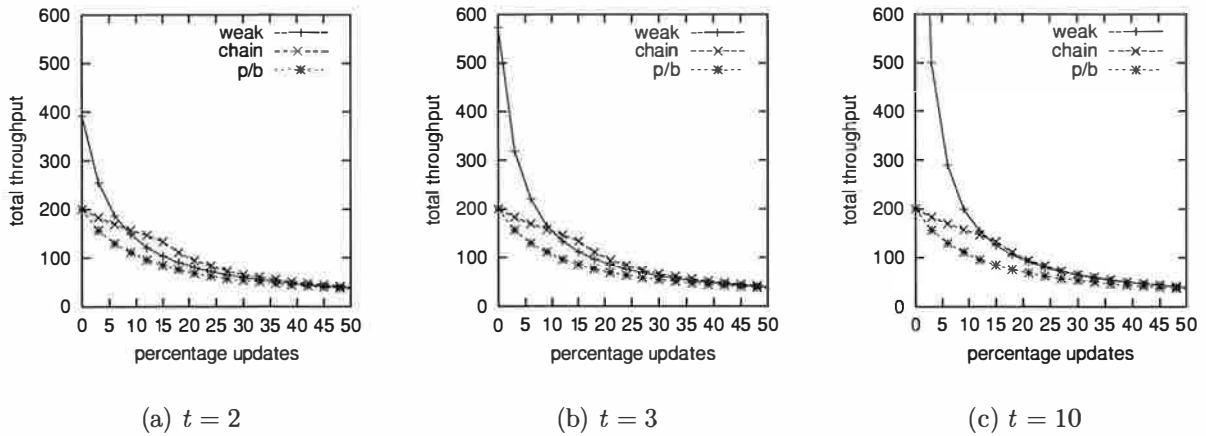


Figure 4: Request throughput as a function of the percentage of updates for various replication management alternatives **chain**, **p/b**, and **weak** (denoting **weak-chain**, and **weak-p/b**) and for replication factors t .

and **weak-p/b** was found to be virtually identical, so Figure 4 has only a single curve—labeled **weak**—rather than separate curves for **weak-chain** and **weak-p/b**.

Observe that chain replication (**chain**) has equal or superior performance to primary-backup (**p/b**) for all percentages of updates and each replication factor investigated. This is consistent with our expectations, because the head and the tail in chain replication share a load that, with the primary/backup approach, is handled solely by the primary.

The curves for the weak variant of chain replication are perhaps surprising, as these weak variants are seen to perform worse than chain replication (with its strong consistency) when there are more than 15% update requests. Two factors are involved:

- The weak variants of chain replication and primary/backup outperform pure chain replication for query-heavy loads by distributing the query load over all servers, an advantage that increases with replication factor.
- Once the percentage of update requests increases, ordinary chain replication outperforms its weak variant—since all updates are done at the head. In particular, under pure chain replication (i) queries are not delayed at the head awaiting completion of update requests (which are relatively time consuming) and (ii) there is more capacity available at the head for update request processing if query requests are not also being handled there.

Since **weak-chain** and **weak-p/b** do not implement strong consistency guarantees, there would seem to

be surprisingly few settings where these replication management schemes would be preferred.

Finally, note that the throughput of both chain replication and primary backup is not affected by replication factor provided there are sufficient concurrent requests so that multiple requests can be pipelined.

5.2 Multiple Chains, No Failures

If each object is managed by a separate chain and objects are large, then adding a new replica could involve considerable delay because of the time required for transferring an object's state to that new replica. If, on the other hand, objects are small, then a large storage service will involve many objects. Each processor in the system is now likely to host servers from multiple chains—the costs of multiplexing the processors and communications channels may become prohibitive. Moreover, the failure of a single processor now affects multiple chains.

A set of objects can always be grouped into a single *volume*, itself something that could be considered an object for purposes of chain replication, so a designer has considerable latitude in deciding object size.

For the next set of experiments, we assume

- a constant number of volumes,
- a hash function maps each object to a volume, hence to a unique chain, and
- each chain comprises servers hosted by processors selected from among those implementing the storage service.

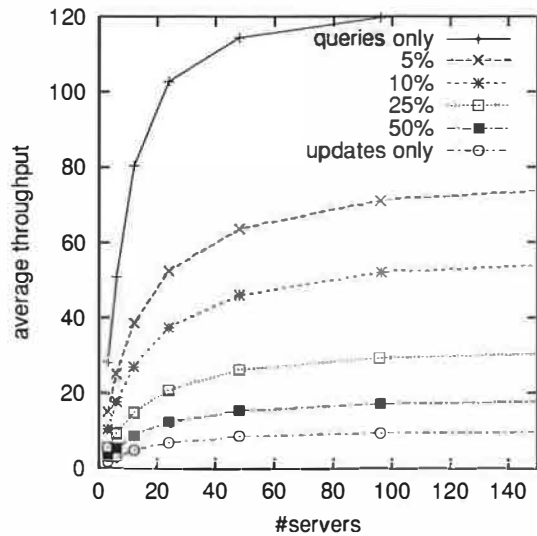


Figure 5: Average request throughput per client as a function of the number of servers for various percentages of updates.

Clients are assumed to send their requests to a *dispatcher* which (i) computes the hash to determine the volume, hence chain, storing the object of concern and then (ii) forwards that request to the corresponding chain. (The master sends configuration information for each volume to the dispatcher, avoiding the need for the master to communicate directly with clients. Interposing a dispatcher adds a 1ms delay to updates and queries, but doesn't affect throughput.) The reply produced by the chain is sent directly to the client and not by way of the dispatcher.

There are 25 clients in our experiments, each submitting queries and updates at random, uniformly distributed over the chains. The clients send requests as fast as they can, subject to the restriction that each client can have only one request outstanding at a time.

To facilitate comparisons with the GFS experiments [11], we assume 5000 volumes each replicated three times, and we vary the number of servers. We found little or no difference among **chain**, **p/b**, **weak chain**, and **weak p/b** alternatives, so Figure 5 shows the average request throughput per client for one-chain replication—as a function of the number of servers, for varying percentages of update requests.

5.3 Effects of Failures on Throughput

With chain replication, each server failure causes a three-stage process to start:

1. Some time (we conservatively assume 10 seconds in our experiments) elapses before the master detects the server failure.
2. The offending server is then deleted from the chain.
3. The master ultimately adds a new server to that chain and initiates a *data recovery* process, which takes time proportional to (i) how much data was being stored on the faulty server and (ii) the available network bandwidth.

Delays in detecting a failure or in deleting a faulty server from a chain can increase request processing latency and can increase transient outage duration. The experiments in this section explore this.

We assume a storage service characterized by the parameters in Table 1; these values are inspired by what is reported for GFS [11]. The assumption about network bandwidth is based on reserving for data recovery at most half the bandwidth in a 100 Mbit/second network; the time to copy the 150 Gigabytes stored on one server is now 6 hours and 40 minutes.

In order to measure the effects of a failures on the storage service, we apply a load. The exact details of the load do not matter greatly. Our experiments use eleven clients. Each client repeatedly chooses a random object, performs an operation, and awaits a reply; a watchdog timer causes the client to start the next loop iteration if 3 seconds elapse and no reply has been received. Ten of the clients exclusively submit query operations; the eleventh client exclusively submits update operations.

<i>parameter</i>	<i>value</i>
number of servers (N)	24
number of volumes	5000
chain length (t)	3
data stored per server	150 Gigabytes
maximum network bandwidth devoted to data recovery to/from any server	6.25 Megabytes/sec
server reboot time after a failure	10 minutes

Table 1: Simulated Storage Service Characteristics.

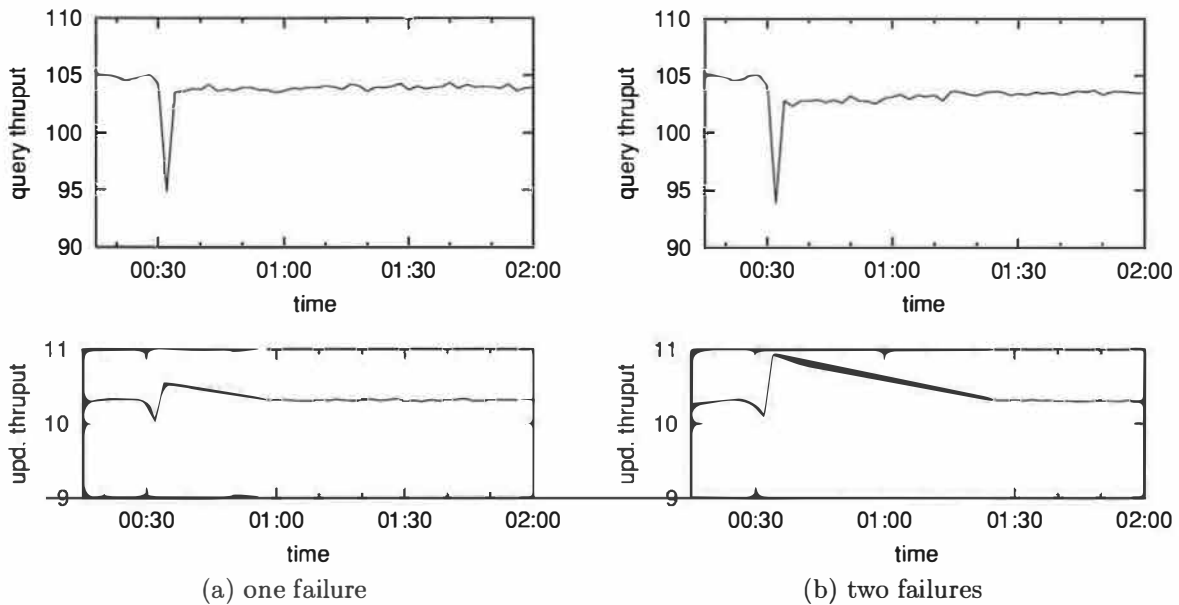


Figure 6: Query and update throughput with one or two failures at time 00:30.

Each experiment described executes for 2 simulated hours. Thirty minutes into the experiment, the failure of one or two servers is simulated (as in the GFS experiments). The master detects that failure and deletes the failed server from all of the chains involving that server. For each chain that was shortened by the failure, the master then selects a new server to add. Data recovery to those servers is started.

Figure 6(a) shows aggregate query and update throughputs as a function of time in the case a single server F fails. Note the sudden drop in throughput when the simulated failure occurs 30 minutes into the experiment. The resolution of the x -axis is too coarse to see that the throughput is actually zero for about 10 seconds after the failure, since the master requires a bit more than 10 seconds to detect the server failure and then delete the failed server from all chains.

With the failed server deleted from all chains, processing now can proceed, albeit at a somewhat lower rate because fewer servers are operational (and the same request processing load must be shared among them) and because data recovery is consuming resources at various servers. Lower curves on the graph reflect this. After 10 minutes, failed server F becomes operational again, and it becomes a possible target for data recovery. Every time data recovery of some volume successfully completes at F , query throughput improves (as seen on the graph).

This is because F , now the tail for another chain, is handling a growing proportion of the query load.

One might expect that after all data recovery concludes, the query throughput would be what it was at the start of the experiment. The reality is more subtle, because volumes are no longer uniformly distributed among the servers. In particular, server F will now participate in fewer chains than other servers but will be the tail of every chain in which it does participate. So the load is no longer well balanced over the servers, and aggregate query throughput is lower.

Update throughput decreases to 0 at the time of the server failure and then, once the master deletes the failed server from all chains, throughput is actually better than it was initially. This throughput improvement occurs because the server failure causes some chains to be length 2 (rather than 3), reducing the amount of work involved in performing an update.

The GFS experiments [11] consider the case where two servers fail, too, so Figure 6(b) depicts this for our chain replication protocol. Recovery is still smooth, although it takes additional time.

5.4 Large Scale Replication of Critical Data

As the number of servers increases, so should the aggregate rate of server failures. If too many servers fail, then a volume might become unavailable. The

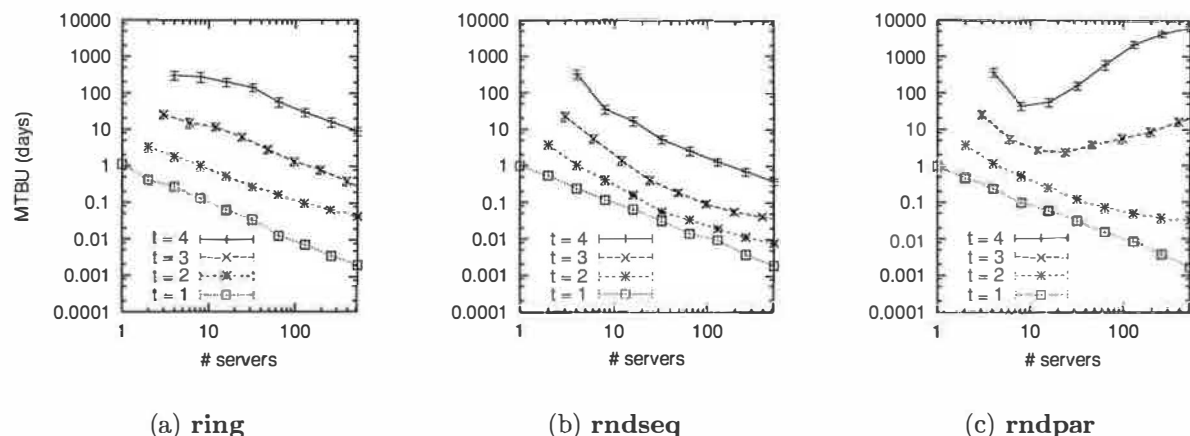


Figure 7: The MTBU and 99% confidence intervals as a function of the number of servers and replication factor for three different placement strategies: (a) DHT-based placement with maximum possible parallel recovery; (b) random placement, but with parallel recovery limited to the same degree as is possible with DHTs; (c) random placement with maximum possible parallel recovery.

probability of this depends on how volumes are placed on servers and, in particular, the extent to which parallelism is possible during data recovery.

We have investigated three volume placement strategies:

- **ring**: Replicas of a volume are placed at consecutive servers on a ring, determined by a consistent hash of the volume identifier. This is the strategy used in CFS [7] and PAST [19]. The number of parallel data recoveries possible is limited by the chain length t .
- **rndpar**: Replicas of a volume are placed randomly on servers. This is essentially the strategy used in GFS.⁵ Notice that, given enough servers, there is no limit on the number of parallel data recoveries possible.
- **rndseq**: Replicas of a volume are placed randomly on servers (as in **rndpar**), but the maximum number of parallel data recoveries is limited by t (as in **ring**). This strategy is not used in any system known to us but is a useful benchmark for quantifying the impacts of placement and parallel recovery.

To understand the advantages of parallel data recovery, consider a server F that fails and was participating in chains C_1, C_2, \dots, C_n . For each chain C_i , data recovery requires a source from which the volume data is fetched and a host that will become the new element of chain C_i . Given enough processors and no constraints on the placement of volumes, it

is easy to ensure that the new elements are all disjoint. And with random placement of volumes, it is likely that the sources will be disjoint as well. With disjoint sources and new elements, data recovery for chains C_1, C_2, \dots, C_n can occur in parallel. And a shorter interval for data recovery of C_1, C_2, \dots, C_n , implies that there is a shorter window of vulnerability during which a small number of concurrent failures would render some volume unavailable.

We seek to quantify the mean time between unavailability (MTBU) of any object as a function of the number of servers and the placement strategy. Each server is assumed to exhibit exponentially distributed failures with a MTBF (Mean Time Between Failures) of 24 hours.⁶ As the number of servers in a storage system increases, so would the number of volumes (otherwise, why add servers). In our experiments, the number of volumes is defined to be 100 times the initial number of servers, with each server storing 100 volumes at time 0.

We postulate that the time it takes to copy all the data from one server to another is four hours, which corresponds to copying 100 Gigabytes across a 100 Mbit/sec network restricted so that only half bandwidth can be used for data recovery. As in the GFS experiments, the maximum number of parallel data recoveries on the network is limited to 40% of the servers, and the minimum transfer time is set to 10 seconds (the time it takes to copy an individual GFS object, which is 64 KBytes).

Figure 7(a) shows that the MTBU for the ring strategy appears to have an approximately Zipfian distribution as a function of the number of servers.

Thus, in order to maintain a particular MTBU, it is necessary to grow chain length t when increasing the number of servers. From the graph, it seems as though chain length needs to be increased as the logarithm of the number of servers.

Figure 7(b) shows the MTBU for **rndseq**. For $t > 1$, **rndseq** has lower MTBU than **ring**. Compared to **ring**, random placement is inferior because with random placement there are more sets of t servers that together store a copy of a chain, and therefore there is a higher probability of a chain getting lost due to failures.

However, random placement makes additional opportunities for parallel recovery possible if there are enough servers. Figure 7(c) shows the MTBU for **rndpar**. For few servers, **rndpar** performs the same as **rndseq**, but the increasing opportunity for parallel recovery with the number of servers improves the MTBU, and eventually **rndpar** outperforms **rndseq**, and more importantly, it outperforms **ring**.

6 Related Work

Scalability. Chain replication is an example of what Jimenéz-Peris and Patiño-Martínez [14] call a ROWAA (read one, write all available) approach. They report that ROWAA approaches provide superior scaling of availability to quorum techniques, claiming that availability of ROWAA approaches improves exponentially with the number of replicas. They also argue that non-ROWAA approaches to replication will necessarily be inferior. Because ROWAA approaches also exhibit better throughput than the best known quorum systems (except for nearly write-only applications) [14], ROWAA would seem to be the better choice for replication in most real settings.

Many file services trade consistency for performance and scalability. Examples include Bayou [17], Ficus [13], Coda [15], and Sprite [5]. Typically, these systems allow continued operation when a network partitions by offering tools to fix inconsistencies semi-automatically. Our chain replication does not offer graceful handling of partitioned operation, trading that instead for supporting all three of: high performance, scalability, and strong consistency.

Large-scale peer-to-peer reliable file systems are a relatively recent avenue of inquiry. OceanStore [6], FARSITE [2], and PAST [19] are examples. Of these, only OceanStore provides strong (in fact, transactional) consistency guarantees.

Google's File System (GFS) [11] is a large-scale cluster-based reliable file system intended for ap-

plications similar to those motivating the invention of chain replication. But in GFS, concurrent overwrites are not serialized and read operations are not synchronized with write operations. Consequently, different replicas can be left in different states, and content returned by read operations may appear to vanish spontaneously from GFS. Such weak semantics imposes a burden on programmers of applications that use GFS.

Availability versus Consistency. Yu and Vahdat [25] explore the trade-off between consistency and availability. They argue that even in relaxed consistency models, it is important to stay as close to strong consistency as possible if availability is to be maintained in the long run. On the other hand, Gray *et al.* [12] argue that systems with strong consistency have unstable behavior when scaled-up, and they propose the *tentative update transaction* for circumventing these scalability problems.

Amza *et al.* [4] present a one-copy serializable transaction protocol that is optimized for replication. As in chain replication, updates are sent to all replicas whereas queries are processed only by replicas known to store all completed updates. (In chain replication, the tail is the one replica known to store all completed updates.) The protocol of [4] performs as well as replication protocols that provide weak consistency, and it scales well in the number of replicas. No analysis is given for behavior in the face of failures.

Replica Placement. Previous work on replica placement has focussed on achieving high throughput and/or low latency rather than on supporting high availability. Acharya and Zdonik [1] advocate locating replicas according to predictions of future accesses (basing those predictions on past accesses). In the Mariposa project [23], a set of rules allows users to specify where to create replicas, whether to move data to the query or the query to the data, where to cache data, and more. Consistency is transactional, but no consideration is given to availability. Wolfson *et al.* consider strategies to optimize database replica placement in order to optimize performance [24]. The OceanStore project also considers replica placement [10, 6] but from the CDN (Content Distribution Network, such as Akamai) perspective of creating as few replicas as possible while supporting certain quality of service guarantees. There is a significant body of work (e.g., [18]) concerned with placement of web page replicas as well, all from the perspective of reducing latency and network load.

Douceur and Wattenhofer investigate how to maximize the worst-case availability of files in FAR-SITE [2], while spreading the storage load evenly across all servers [8, 9]. Servers are assumed to have varying availabilities. The algorithms they consider repeatedly swap files between machines if doing so improves file availability. The results are of a theoretical nature for simple scenarios; it is unclear how well these algorithms will work in a realistic storage system.

7 Concluding Remarks

Chain replication supports high throughput for query and update requests, high availability of data objects, and strong consistency guarantees. This is possible, in part, because storage services built using chain replication can and do exhibit transient outages but clients cannot distinguish such outages from lost messages. Thus, the transient outages that chain replication introduces do not expose clients to new failure modes—chain replication represents an interesting balance between what failures it hides from clients and what failures it doesn't.

When chain replication is employed, high availability of data objects comes from carefully selecting a strategy for placement of volume replicas on servers. Our experiments demonstrated that with DHT-based placement strategies, availability is unlikely to scale with increases in the numbers of servers; but we also demonstrated that random placement of volumes does permit availability to scale with the number of servers if this placement strategy is used in concert with parallel data recovery, as introduced for GFS.

Our current prototype is intended primarily for use in relatively homogeneous LAN clusters. Were our prototype to be deployed in a heterogeneous wide-area setting, then uniform random placement of volume replicas would no longer make sense. Instead, replica placement would have to depend on access patterns, network proximity, and observed host reliability. Protocols to re-order the elements of a chain would likely become crucial in order to control load imbalances.

Our prototype chain replication implementation consists of 1500 lines of Java code, plus another 2300 lines of Java code for a Paxos library. The chain replication protocols are structured as a library that makes upcalls to a storage service (or other application). The experiments in this paper assumed a “null service” on a simulated network. But the library also runs over the Java socket library, so it

could be used to support a variety of storage service-like applications.

Acknowledgements. Thanks to our colleagues Håkon Brugård, Kjetil Jacobsen, and Knut Omang at FAST who first brought this problem to our attention. Discussion with Mark Linderman and Sarah Chung were helpful in revising an earlier version of this paper. We are also grateful for the comments of the OSDI reviewers and shepherd Margo Seltzer. A grant from the Research Council of Norway to FAST ASA is noted and acknowledged.

Van Renesse and Schneider are supported, in part, by AFOSR grant F49620-03-1-0156 and DARPA/AFRL-IFGA grant F30602-99-1-0532, although the views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

Notes

¹The case where $V = \text{newVal}$ yields a semantics for update that is simply a file system write operation; the case where $V = F(\text{newVal}, \text{objID})$ amounts to support for atomic read-modify-write operations on objects. Though powerful, this semantics falls short of supporting transactions, which would allow a request to query and/or update multiple objects indivisibly.

²An actual implementation would probably store the current value of the object rather than storing the sequence of updates that produces this current value. We employ a sequence of updates representation here because it simplifies the task of arguing that strong consistency guarantees hold.

³If $\text{Hist}_{\text{objID}}$ stores the current value of objID rather than its entire history then “ $\text{Hist}_{\text{objID}} \cdot r$ ” should be interpreted to denote applying the update to the object.

⁴If $\text{Hist}_{\text{objID}}^i$ is the current state rather than a sequence of updates, then \preceq is defined to be the “prior value” relation rather than the “prefix of” relation.

⁵Actually, the placement strategy is not discussed in [11]. GFS does some load balancing that results in an approximately even load across the servers, and in our simulations we expect that random placement is a good approximation of this strategy.

⁶An unrealistically short MTBF was selected here to facilitate running long-duration simulations.

References

- [1] S. Acharya and S.B. Zdonik. An efficient scheme for dynamic data replication. Technical Report CS-93-43, Brown University, September 1993.
- [2] A. Adya, W.J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J.R. Douceur, J. Howell, J.R. Lorch, M. Theimer, and R.P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proc. of the 5th Symp. on Operating Systems Design and Implementation*, Boston, MA, December 2002. USENIX.
- [3] P.A. Alsberg and J.D. Day. A principle for resilient sharing of distributed resources. In *Proc. of the 2nd Int. Conf. on Software Engineering*, pages 627–644, October 1976.
- [4] C. Amza, A.L. Cox, and W. Zwaenepoel. Distributed Versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *Proc. of Middleware'03*, pages 282–304, Rio de Janeiro, Brazil, June 2003.
- [5] M.G. Baker and J.K. Ousterhout. Availability in the Sprite distributed file system. *Operating Systems Review*, 25(2):95–98, April 1991. Also appeared in the 4th ACM SIGOPS European Workshop – Fault Tolerance Support in Distributed Systems.
- [6] Y. Chen, R.H. Katz, and J. Kubiawicz. Dynamic replica placement for scalable content delivery. In *Proc. of the 1st Int. Workshop on Peer-To-Peer Systems*, Cambridge, MA, March 2002.
- [7] F. Dabek, M.F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of the 18th ACM Symp. on Operating Systems Principles*, Banff, Canada, October 2001.
- [8] J.R. Douceur and R.P. Wattenhofer. Competitive hill-climbing strategies for replica placement in a distributed file system. In *Proc. of the 15th International Symposium on Distributed Computing*, Lisbon, Portugal, October 2001.
- [9] J.R. Douceur and R.P. Wattenhofer. Optimizing file availability in a secure serverless distributed file system. In *Proc. of the 20th Symp. on Reliable Distributed Systems*. IEEE, 2001.
- [10] D. Geels and J. Kubiawicz. Replica management should be a game. In *Proc. of the 10th European SIGOPS Workshop*, Saint-Emilion, France, September 2002. ACM.
- [11] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proc. of the 19th ACM Symp. on Operating Systems Principles*, Bolton Landing, NY, October 2003.
- [12] J. Gray, P. Helland, P. O’Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of the International Conference on Management of Data (SIGMOD)*, pages 173–182. ACM, June 1996.
- [13] J.S. Heidemann and G.J. Popek. File system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.
- [14] R. Jimenéz-Peris and M. Patiño-Martínez. Are quorums an alternative for data replication? *ACM Transactions on Database Systems*, 28(3):257–294, September 2003.
- [15] J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system (preliminary version). *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.
- [16] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.
- [17] K. Petersen, M.J. Spreitzer, D.B. Terry, M.M. Theimer, and A.J. Demers. Flexible update propagation for weakly consistent replication. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, pages 288–301, Saint-Malo, France, October 1997.
- [18] L. Qiu, V.N. Padmanabhan, and G.M. Voelker. On the placement of web server replicas. In *Proc. of the 20th INFOCOM*, Anchorage, AK, March 2001. IEEE.
- [19] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large scale, persistent peer-to-peer storage utility. In *Proc. of the 18th ACM Symp. on Operating Systems Principles*, Banff, Canada, October 2001.
- [20] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.
- [21] F.B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems*, 2(2):145–154, May 1984.
- [22] F.B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.
- [23] M. Stonebraker, P.M. Aoki, R. Devine, W. Litwin, and M. Olson. Mariposa: A new architecture for distributed data. In *Proc. of the 10th Int. Conf. on Data Engineering*, Houston, TX, 1994.
- [24] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Computer Systems*, 22(2):255–314, June 1997.
- [25] H. Yu and A. Vahdat. The cost and limits of availability for replicated services. In *Proc. of the 18th ACM Symp. on Operating Systems Principles*, Banff, Canada, October 2001.

Boxwood: Abstractions as the Foundation for Storage Infrastructure

John MacCormick, Nick Murphy, Marc Najork,
Chandramohan A. Thekkath, and Lidong Zhou

Microsoft Research Silicon Valley

Abstract

Writers of complex storage applications such as distributed file systems and databases are faced with the challenges of building complex abstractions over simple storage devices like disks. These challenges are exacerbated due to the additional requirements for fault-tolerance and scaling. This paper explores the premise that high-level, fault-tolerant abstractions supported directly by the storage infrastructure can ameliorate these problems. We have built a system called Boxwood to explore the feasibility and utility of providing high-level abstractions or data structures as the fundamental storage infrastructure. Boxwood currently runs on a small cluster of eight machines. The Boxwood abstractions perform very close to the limits imposed by the processor, disk, and the native networking subsystem. Using these abstractions directly, we have implemented an NFSv2 file service that demonstrates the promise of our approach.

1 Introduction

Implementing distributed, reliable, storage-intensive software such as file systems or database systems is hard. These systems have to deal with several challenges including: matching user abstractions (e.g., files, directories, tables, and indices) with those provided by the underlying storage, designing suitable data placement, prefetching, and caching policies, as well as providing adequate fault-tolerance, incremental scalability, and ease of management. Indeed, it is generally believed that building a distributed file system or a distributed database with all these properties is an unrealistic ideal. Our hypothesis in this paper is that this perceived difficulty can be considerably lessened through the use of suitable abstractions such as trees, linked lists, and hash-tables, provided directly by the storage subsystem, without compromising performance, scalability, or the manageability of the storage system or the higher-level subsystems built

on top of it. We have built a system called Boxwood to explore the feasibility and utility of providing such high-level abstractions or data structures as the fundamental storage infrastructure. Using these abstractions directly, we have implemented a highly available and scalable NFSv2 server that runs on multiple machines, coherently exporting the same underlying file system.

Although Boxwood's approach to storage is a significant departure from traditional block-oriented interfaces provided by disks—whether physical, logical [17], or virtual [21]—we think it provides some key advantages. One advantage, as evidenced by our experience with the multi-machine NFS server, is that by directly integrating data structures into the persistent storage architecture, higher-level applications are simpler to build, while getting the benefits of fault-tolerance, distribution, and scalability at little cost. Furthermore, abstractions that can inherently deal with sparse and non-contiguous storage free higher level software from dealing with address-space or free-space management. In contrast, even sophisticated virtual disk systems that provide scalability and ease of management require higher layers like the file system to deal with free space management, data placement, and maintaining user-visible abstractions [32]. A third advantage is that using the structural information inherent in the data abstraction can allow the system to perform better load-balancing, data prefetching, and informed caching. These mechanisms can be implemented once in the infrastructure instead of having to be duplicated in each subsystem or application.

Our earliest experience with Boxwood convinced us that there is no single universal abstraction to storage that will serve the needs of all clients. Our current prototype provides two: a *B-tree* abstraction, which allows typical operations like lookups, insertions, deletions, and enumerations, and a simpler *chunk store*, where variable sized data items can be allocated, freed, written, and read, in much the same way that memory is today.

Our specific choices were motivated by several obser-

ventions. First, B-trees are a very useful abstraction for many storage needs found in file systems, databases and the like. Second, building a high-performance, scalable and *distributed* B-tree is a considerable challenge (even building a centralized version with good concurrent performance is difficult). We believe that our experience with this data structure will complement the existing literature in building distributed data structures like hash tables. Third, we believe that our simple chunk store abstraction provides a better match for applications that do not need the strict atomicity guarantees or the rigid structure of a B-tree. This simpler abstraction offers good performance and much flexibility to client applications while offloading the details of free space (or in a virtual disk environment, address space) management.

Our prototype is implemented by a collection of “server nodes”, each containing a CPU, RAM, one or more disks, and a network interface packaged as a rack-mounted server. One can imagine alternative implementations of server nodes ranging from individual disk units to disk controllers that control sets of disks.

In addition to its focus on distributed storage abstractions, this paper also offers some insights into the structure of fault-tolerant distributed systems. The classic approach to building such systems is to use Lamport’s replicated state machines with Paxos [20]. Our approach, although highly reliant on Paxos for consensus, uses a fault-tolerant distributed lock service and the simple notion of shared memory (or more precisely shared store) programming to deal with the inherent complexity of a distributed system with independent failures. We believe the lessons learned may be valuable in the design of other fault-tolerant distributed systems.

2 Boxwood System Structure

The overall goal of the Boxwood project is to experiment with data abstractions as the underlying basis for storage infrastructure. Generally, the term storage infrastructure connotes several requirements, a few of which are: *redundancy and backup schemes* to tolerate failures, *expansion mechanisms* for load and capacity balancing, and *consistency maintenance* in the presence of failures. Thus, ideally Boxwood needs to go well beyond providing distributed data structures. Our current status does not satisfy this ideal, but we have made much progress. For example, to deal with transient failures, we provide services for logging and transaction recovery. To deal with the correctness of replication in the presence of failures, automatic reconfiguration and expansion, we provide mechanisms (e.g., a Paxos consensus module, a lock service, and a failure detector) to insure a correct inventory of the components in the system and to provide a consistent view of the overall system.

In this section, we describe parts of our design as it relates to data abstractions and storage infrastructure mechanisms. We envisage our system being deployed in a machine room or in an enterprise cluster as the principal storage infrastructure used by file systems, database systems, and other services. This environment justifies several assumptions that impact our design choices. We first enumerate these assumptions and design principles before describing our system in greater detail.

2.1 Preliminaries

The Boxwood system is targeted at an environment that has multiple processing nodes each with locally attached storage, interconnected by a high-speed network. These processors run the Boxwood software components to implement abstractions and other services. Software running on a processor communicates with locally attached disks using a low-level interface similar to the UNIX raw device interface. We use remote procedure call (RPC) to access resources and services executing on remote processors.

The Boxwood system is organized as several interdependent services. We use layering as a way of managing the complexity in a Boxwood system. For example, the B-tree and the chunk store services mentioned earlier in Section 1 are constructed by layering the former on top of the latter. The chunk store service, in turn, is layered on top of a simple *replicated logical device* abstraction (to be described in Section 3.4). Although layering has the potential for reducing performance because of context switching overheads, our design avoids these problems by running all layers within a single address space.

Our interconnection network is Gigabit Ethernet; we therefore feel justified in providing fault-tolerance by *synchronous replication* of data on two disks attached to separate machines. With this scheme, under fault-free operations, the primary replica must wait for the secondary to finish writing its copy before it can return to the client. This wait can be large on a slow network, but is tolerable in a high-speed LAN. We also feel justified in assuming that the cost of making RPCs is small and that the network can be scaled by adding more switches. Our implementation results bear out these assumptions.

We use a security model that is appropriate to the target environment. Specifically, we assume that the machines are within a single administrative domain and are physically secure. We therefore send messages between machines in the clear and make no special provisions for encryption, authentication, or security.

We assume that CPUs, disks, and networks can fail. Such failures can be transient or permanent. Examples of transient failures that we can tolerate are: a faulty

power supply takes down a machine (and its attached disks), which will come back up without the contents of its RAM after power is restored; or the operator mistakenly unplugs a network cable. Examples of permanent failures are: a disk suffers catastrophic media failure, or a server's log is destroyed beyond repair. We assume, realistically, that the failure of a disk affects only that disk, but the failure of a machine affects it and all the disks attached to it. Although unlikely, we assume that networks can partition. Failures are assumed to be fail-stop.

Our fundamental mechanism for protecting data against catastrophic media failure is chained-declustered replication [14]. Thus, the permanent failure of a single disk will not cause data loss or data unavailability. In fact, chained-declustering prevents data loss even in the presence of many combinations of multiple disk failures as well, but not against all combinations of multiple disk failures. We also deal with the failure mode when all machines suffer a transient power outage and come back having lost the contents of RAM.

In our design, each service consists of software modules executing on multiple machines. Each service independently arranges for failover and high availability in the presence of multiple failures. For instance, our Paxos consensus service (described in Section 3.1) works as long as a majority of Paxos servers are running. Thus, double failures can be tolerated by running five Paxos servers and triple failures with seven. Similarly, our lock service (described in Section 3.3) uses a single master and one or more slaves as standby. Depending on the number of slaves we choose to run, we can tolerate multiple permanent failures. Our B-tree and NFS services described in Sections 3.7 and 5 impose no additional availability constraints as long as at least one instance of each module is running and the underlying services (e.g., locking, consensus, and replicated data) are available.

As the scale of the deployment increases, the probability of multiple failures increases. Our design is most vulnerable to increased disk failures in this regard. If the probability of double disk failures becomes a serious problem, we can use a different data protection scheme (e.g., triplexing or erasure coding) at the lowest layers without changing the design of any of the other services.

The principal client-visible abstractions that Boxwood provides are a B-tree abstraction and a simple chunk store abstraction provided by the *Chunk Manager*. Figure 1 shows the organization of these abstractions relative to each other. We introduce them briefly here, but defer a fuller description to later sections.

2.2 B-tree Abstraction

B-trees and their variants are widely viewed as the best general-purpose data structure for implementing a dictio-

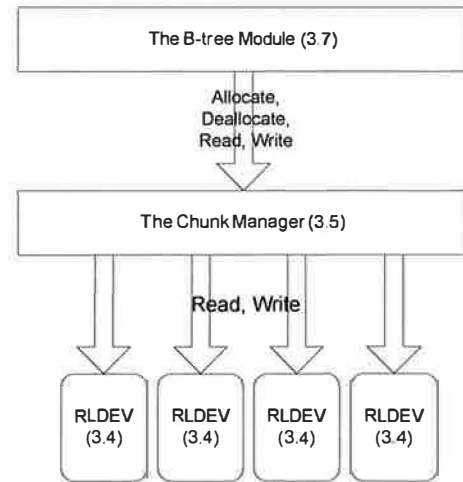


Figure 1: **Design of the Boxwood abstractions.** The B-tree is layered atop the chunk manager, which is layered on top of the replicated logical device. The numbers in parentheses refer to the section describing the design of the module.

nary (supporting insertion, lookup, deletion, and enumeration of key-value pairs) on secondary storage. B-trees are also complex enough to exercise fully the features and foibles of a distributed storage architecture. Therefore, they seemed an excellent candidate for the first data abstraction to be implemented within Boxwood.

The Boxwood B-tree module is a distributed implementation of Sagiv's [28] B-link tree algorithm, which is a variant of the Lehman-Yao B-link tree algorithm [22]. Sagiv's algorithm has the desirable property that locking is considerably simplified from traditional B-tree algorithms without compromising concurrency or atomicity. Sagiv's original B-link tree algorithm (like its classic B -tree, B^+ -tree, or B^* -tree counterparts) runs in a single machine environment, uses locks for synchronizing accesses amongst multiple threads, and stores data either in memory or persistently on disk. Sagiv's algorithm is well suited for a distributed implementation, an observation independently made by Johnson and Colbrook [16].

Since the algorithm to implement the B-link tree can already deal with thread concurrency within a single machine, our design extends this design to multiple machines by ensuring two simple constraints are met:

- Threads executing on multiple machines use global locks for synchronizing access to shared data.
- Data stored by one thread running on a machine can be accessed by another thread on any other.

The first constraint is readily provided by our distributed lock service described in Section 3.3. To meet the second

requirement, we could use an existing virtual (or logical) disk or logical volume manager, but we decided against this for the following reasons. Existing logical/virtual disk systems would still require us to do our own management of physical/virtual space. Most systems we know of did not support our needs for fault-tolerance, incremental expansion, and scalability. The few systems that do (e.g., Petal [21] or FAB [9]), implement their own logging and recovery schemes, which duplicate much of the logging and recovery required at the B-link tree level, increasing our bookkeeping overheads, and making it difficult to implement certain optimizations that were possible in our design.

2.3 Chunk Data Store

The data store used by the B-tree abstraction is provided by a *chunk manager*. The principal function of the chunk manager is to hide the details of the physical storage media and to provide a level of address mapping so that the B-tree algorithm can deal with opaque pointers to stored data. The chunk manager acts much like a memory allocator, in that it hands out variable length chunks of the data store that can subsequently be written to, read from, or deallocated.

The chunk manager carves out chunks of storage by using the services of a lower layer called the *replicated logical device (RLDev)* layer. Each RLDev provides access to a fixed amount of chain-declustered storage. An RLDev is implemented on two machines using two separate physical disk drives for replication.

2.4 Infrastructure Services

In addition to the software modules that implement the various abstractions, Boxwood contains three important modules that provide essential distributed system services. These are heavily used within the Boxwood system to implement the abstractions, and can also be used directly by the external clients of the Boxwood system. These services are:

- **Paxos service.** This is an implementation of the Paxos part-time parliament algorithm [20]. It is used to store global system state such as the number of machines and the number of RLDevs in the system. It is also used by the distributed lock service to keep track of client information and the identity of the lock master for recovery when the lock service has a transient failure.
- **Lock Service.** This provides a distributed lock service that handles multiple-reader, single-writer locks. This service is used by the RLDev module, the chunk manager, and our multi-node NFS server.

- **Transaction Service.** This service provides a redo-undo logging facility and transaction support for the recovery of the B-tree module and in the NFS server.

3 Boxwood Design

This section describes the design of the various components of the Boxwood system in more detail. Since there are several interdependencies amongst these components, we describe them in an order that minimizes forward references.

3.1 Paxos Service

The Paxos service is a “general-purpose” implementation of the state machine approach using the Paxos part-time parliament algorithm. We refer to our service as general-purpose because clients of the service can define arbitrary client-specific state and pass client-specific “decrees” to modify and query this state by making RPC calls to the Paxos service. The state maintained by the Paxos service is replicated on a collection of independent machines. The Paxos algorithm provably guarantees that the state changes occur in the same order on each replica and that the state is available and is consistent as long as a majority of these replicas are non-faulty.

Boxwood depends on three different types of client states maintained within Paxos. Each client state typically refers to the essential state required by an internal Boxwood service such as the lock service, or the RLDev layer, or the chunk manager. This state is consulted by each service as appropriate to perform recovery or reconfiguration of that layer. Typically, the state includes the machines, disks, and other resources like network port identifiers used by the client.

The Paxos service is implemented on a small collection of machines, typically three, with two machines constituting a quorum or majority. We do not dedicate an entire machine to implementing the service; the Paxos service instance on a machine is restricted to a single server process. Paxos state is maintained on disks that are locally attached to the machines hosting the server. We do not rely on these disks to be fault tolerant, but merely that they are persistent. The failure of the disk storage or the machine is considered as a failure of the Paxos server. Our choice of three machines is arbitrary; it allows us to tolerate the failure of one machine in our small cluster. We can tolerate the failure of k machines by running the service on $2k + 1$ machines.

We draw a clear distinction between the characteristics of the Paxos service and the characteristics of the rest of the storage- and abstraction-related services. We have isolated the scaling of the overall system from the

scaling of Paxos. Client state stored on Paxos can be dynamically changed with the addition of new RLDevs, new chunk managers, new machines, or new disks. This does not require us to dynamically change the number of Paxos servers that store the state. It is conceivable that at extremely large scales, we might wish to increase on the fly the number of Paxos servers in the system to guard against increased machine failures. If so, we will need to implement the protocols necessary to increase or decrease dynamically the number of Paxos servers *per se*.

We ensure that Paxos is only involved when there are failures in the system or there are reconfigurations of the system. This allows us to avoid overloading the servers for the common case operations such as reads and writes, which need to complete quickly.

We implement a slightly restricted form of Paxos by decreasing the degree of concurrency allowed. In standard Paxos, multiple decrees can be concurrently executed. In our system, we restrict decrees to be passed sequentially. This makes the implementation slightly easier without sacrificing the effectiveness of the protocol for our purposes.

To ensure liveness properties in a consensus algorithm like Paxos, it can be shown that it is only necessary to use a failure detector with fairly weak properties [5]. In principle, we too only need such a weak failure detector. However, we need failure detectors with stronger guarantees for our RPC and lock server modules. Rather than implement multiple failure detection modules, we use a single one with more restrictions than those required by Paxos.

3.2 Failure Detector

Our failure detection module is implemented by having machines exchange periodic *keepalive* beacons. Each machine is monitored by a collection of *observer* machines with which it exchanges keepalive messages. A machine can check on the status of any machine by querying the observers. A machine is considered failed only when a majority of the observer machines have not heard keepalive messages from it for some threshold period. The invariants we maintain are that, (a) if a machine dies, the failure detector will eventually detect it, and that, (b) if the failure detector *service* (not to be confused with an individual observer) tells a client that a machine is dead, then that machine is indeed dead.

Figure 2 sketches the message protocol assuming a single observer, rather than a majority. Our messages are sent using UDP and may fail to arrive or arrive out of order, albeit with very low probability on a LAN. We do not assume a synchronized clock on each machine, but we do assume that the clocks go forwards, the clock drift on the machines is bounded, and UDP message delays

are non-zero.

A client (Client A in the figure) periodically (at intervals of ΔT) sends out beacon messages to the observer. These messages may or may not be delivered reliably. The observer echoes each beacon message it receives back to the client. At any point in time, the observer considers a client dead if it has not received a beacon from the client in the preceding *GracePeriod*. The client considers the receipt of the echo as an acknowledgement from the observer. The client keeps track of the last time it sent a beacon that was acknowledged by the observer. If more than *GracePeriod* time elapses without an acknowledgement, it considers itself dead and kills itself.

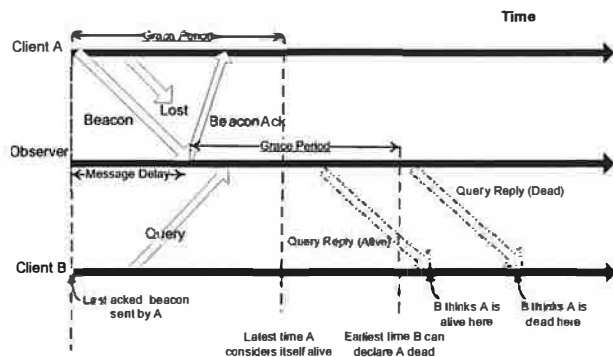


Figure 2: Message protocol for the failure detector assuming a single observer and two clients. Time advances to the right. If B thinks A is dead, then A must be dead.

Another client (Client B in the figure) that wishes to monitor the first client (Client A) sends a query message to the observer. The observer then sends B its view about A. If B receives a reply from the observer claiming A is dead, it considers A dead; otherwise it considers A alive. Given our assumptions about clock drift and non-zero message delay, this protocol is conservative and maintains our invariants.

In reality, we don't use a single observer, but use a collection of observers for reasons of fault-tolerance. In this case, client B in the figure pronounces A dead only if it gets replies from a majority of observers (instead of the observer) that all pronounce A dead. If B does not receive a majority of responses that pronounce A dead, it assumes that A is alive. A, on the other hand, considers itself dead as soon as *GracePeriod* time elapses without an acknowledgement from a majority. This protocol can lead to a state where A considers itself dead, while B thinks A is alive. But more to the point, if B thinks A is dead (because a majority pronounces A dead), then A cannot have received an acknowledgement from a majority, and will consider itself dead. This maintains our invariant that if B considers A dead (because the failure detector said so), then A must be dead.

Our protocol, as sketched, only works if clients that die do not get resurrected later and start sending beacons. We ensure this in practice by having each client use a monotonically increasing incarnation number each time it has a transient failure.

The values of `GracePeriod`, ΔT , and the number of observers are tunable parameters. We have found one second, 200 ms, and three to be suitable in our environment. We use the same observer machines for all the clients in the system for convenience, although it is feasible to use different subsets of machines as observers for each specific client.

3.3 Distributed Lock Service

The lock service provides a simple general purpose locking mechanism that allows its clients to acquire multiple-reader single-writer locks. Our design borrows techniques used in earlier work [4, 6, 12, 32]; we describe the details of our scheme to underscore our rationale for particular choices.

Locks are uniquely identified by byte arrays on which the lock service does not impose any semantics. Although locks have no explicit timeouts associated with them, the failure detector is used to time out unresponsive clients. So in essence, our locks act as degenerate leases [12].

Clients of the lock service have a clerk module linked into their address spaces. Leases are cached by the clerk and are only revoked by the service if there is a conflicting request by another clerk. A clerk blocks an incoming revocation request until all currently outstanding local uses of the lease have completed. An optimization, which we don't currently support, is for the clerk to release a lease when it has not used it for some time.

Clients can optionally arrange with the lock service to call a recovery function on another instance of the same client if the first instance were to fail. The lock service guarantees that the leases acquired by a client that has subsequently failed will not be released until this recovery is successfully completed. This is a modest extension to Gray and Cheriton's standard lease mechanism (which primarily focused on write-through client caching) to deal with residual state that exists in a client after the lease has timed out. An example of such usage can be found in the B-tree module described in Section 3.7.

The failure of a lock client is determined by the failure detector. Notice that for our scheme to work correctly, both the client and the lock service must use the failure detector in a consistent fashion. Otherwise, the lock service could revoke the lease, while the client believed it had the lease. We ensure correct behavior by requiring two conditions of our failure detector. First, if a client

dies, then the failure detector will eventually notice that it is dead. Second, if the failure detector claims that a client is dead, then the client must have died some time prior (but perhaps is alive now if it was a transient failure). We also ensure that a client that comes alive and finishes recovery always assumes that it holds no leases and registers with the lock service.

For fault-tolerance, the lock service consists of a single master server and one or more slave servers running on separate machines. In our cluster, we typically use only a single slave server, but if multiple machine failures are common, additional slaves can be used. Only the master server, whose identity is part of the global state in Paxos, hands out leases. The lock service also keeps the list of clerks as part of its Paxos state.

If the failure detector pronounces the current master dead, the slave takes over after passing a Paxos decree that changes the identity of the current master. The new master recovers the lease state by first reading Paxos state to get a list of clerks. Then it queries the clerks for their lease state. It is possible that some of the clerks are dead at this point. In this case, the lock service calls recovery on behalf of these clients on the clients that are alive, and considers all leases held by the dead clients as free. If no recovery procedure has been established for a dead client, the lock service considers all leases held by that client as free.

Lock service clerks query Paxos to determine the identity of the master server. This information is cached until an RPC to the currently master returns an exception, at which point it is refreshed. RPCs to a machine return an exception if the failure detector claims the target is dead.

The lock service fails if all (both) servers fail. Clients cannot make forward progress until it is re-established.

We use a simple master-slave design for the lock service because we believe other more elaborate, scalable schemes are unnecessary in most storage-centric environments. Our rationale is that even in elaborate schemes with several active lock servers, a specific lock will be implemented by a single lock server at any given time. If this is a highly contended lock, then lock contention due to data sharing becomes a performance problem on the clients long before the implementation of the lock server itself becomes a bottleneck. Our experience with deploying Petal/Frangipani, which had a more scalable lock service, seems to bear this out.

3.4 RLDevs: Replicated, Logical Devices

Boxwood implements storage replication through a simple abstraction we call a *replicated, logical device* (RLDev). An RLDev is logically a block device in that it expects block-aligned accesses in multiples of block units.

We chose to implement replication at a fairly low level in the abstraction hierarchy for two principal reasons. First, by providing replication at a low level, all higher layers, which are typically more complex in nature, can depend on fault-tolerant storage, which makes the logic of the higher layers simpler to reason about. For instance, our implementation of the chunk manager (to be described in Section 3.5) was considerably easier because of the RLDev layer. Second, by replicating at a low level of abstraction, the relevant replication, mapping, and failover logic, as well as internal data structures, can be made simple.

RLDevs implement chained declustering. A single RLDev is of fixed size and consists of two *segments* of equal size located on disks on two different machines. A single disk will contain segments from multiple RLDevs. The list of RLDevs, the segments belonging to them, the identity of machines that host the primary and the secondary segments, and the disks are all part of the global state maintained in Paxos. If an RLDev is added or if the locations of the segments belonging to an RLDev are changed, a Paxos decree must be passed.

The replication protocol is fairly standard. One replica is designated the primary, and the other the secondary. On initialization, a replica reads its state from the Paxos service and monitors its peer using the failure detector. When both replicas are up, writes are performed on both and reads on either. A client sends write requests to the primary, which forwards the request to the secondary and waits for completion.

Clients of the RLDev use hints to determine where the replicas are located. Hints can sometimes be wrong and can be updated by reading the Paxos state. An RLDev clerk linked in with the clients handles the details of dealing with hints and refreshing them as appropriate.

When one of the replicas dies, the surviving replica continues to accept writes (and reads). We call these *degraded mode* writes because the system is accepting new data but not replicating it on the dead replica. A subsequent failure of the surviving replica (the one that has accepted the degraded mode data) before the first replica has finished recovering will lead to data loss. Before it accepts these “degraded mode” writes, the survivor passes a decree to that effect so that if the dead replica were to come back after a transient failure, it knows to reconcile its stale data, and more importantly, not to accept new reads or writes if it cannot reconcile its stale data. This can happen because the replica that was working in “degraded mode” now happens to be dead. All blocks that have degraded mode writes on them are put in a log (called the *degraded mode log*) so that reconciliation is fast and only involves the affected blocks.

Notice that in the worst case, the entire segment could have been written in degraded mode. Thus, when a pre-

viously dead replica comes up, we have to be prepared to copy the entire segment. We can leverage this mechanism to implement the automatic *reconfiguration* of an RLDev. By a reconfiguration operation, we mean the redistribution of the data in an RLDev to a different disk and/or machine to enable load balancing. Making an RLDev relatively small makes it easy to copy its data quickly on a high-bandwidth LAN link, thereby cutting down reconfiguration time.

As mentioned previously, when both replicas are up, the primary waits for the secondary to commit the write before returning to the client. In order to cope with a transient failure when the writes are in flight, an RLDev implements a *dirty region log*, which serves a different purpose than the degraded mode log mentioned previously. The dirty region log on the primary keeps track of writes that are in flight to the secondary. When the secondary replies, the log entry can be removed from the primary in principle. To recover from a transient failure, the primary consults its dirty region log to determine the writes that were in flight and sends the secondary the current contents of its disks for these writes.

In cases where the client can deal with the two replicas differing after a crash, RLDevs allow the client to turn off the dirty region log. Such clients must explicitly read and write each replica to reconcile the differences after a crash. In Boxwood, all such clients already maintain a log for other reasons, and there is no added cost for maintaining the equivalent of the dirty region information, except a modest violation in layering. We felt this tradeoff was justified for the performance gain of saving an additional disk write.

Recovering an RLDev is fairly straightforward. There are two failure cases: a permanent failure of a disk or the transient failure of processor and its attached disks. When there is a permanent disk failure, the RLDevs that are hosted on that disk must be reconstituted on a new (or perhaps more than one) disk, but the recovery of each RLDev proceeds independently. An RLDev recovering from a permanent disk failure contacts the RLDev on the machine that hosts its surviving segment and copies the contents of the entire segment. In contrast, if the failure was transient, then the data retrieved from the peer is limited to any degraded mode writes the peer has for the recovering segment. After degraded mode writes have been applied on the recovering segment, the dirty region log of the surviving segment is read and sent to the recovering server to apply any in-flight writes. When the recovering segment is up to date, the recovering server passes another Paxos decree indicating that the state of the RLDev is normal and stops recording degraded mode writes.

3.5 The Chunk Manager

The fundamental storage unit in Boxwood is the *chunk*. A chunk is a sector-aligned sequence of consecutive bytes on an RLDev, allocated in response to a single request. Every chunk is identified by an *opaque handle* that is globally unique in the system.

Chunks are managed by the chunk manager module, which supports four operations: *allocate*, *deallocate*, *read*, and *write*. Deallocated handles are guaranteed never to be reused. Reading or writing an unallocated handle raises an exception.

Since an RLDev can be accessed by any machine with a suitable clerk linked in, it is possible, in principle, to have any chunk manager allocate chunks from any RLDev. In fact, as long as a single chunk manager is alive, we can manage all the RLDevs that are non-faulty. However, for simplicity, ease of load balancing, and performance, we designate a pair of chunk managers running on two different machines to manage space from a given set of RLDevs. Typically, these RLDevs will have their primary and secondary segments located on disks that are local to the two chunk managers. This reduces the number of network hops required to perform chunk operations. One of the pair of chunk managers acts as the primary initiating all allocations and deallocations, while either can perform reads and writes.

The mapping between the opaque handle and the RLDev offset is replicated persistently on RLDevs. This mapping information is accessed often: an allocate call requires a new mapping to be created; a deallocate call deletes the mapping; reads and writes require this mapping to be consulted. We therefore cache the mapping on both the primary and the secondary.

A *map lock* from the lock service protects mappings. Only the primary makes changes to the mappings; this lock is therefore always cached on the primary in exclusive mode. When the primary changes the mappings, it writes the new mapping to the RLDev and sends an RPC to the secondary, which directly updates its cached copy of the mappings without acquiring the lock.

On startup, the secondary has to read the latest mapping state from the RLDev so that subsequent RPCs from the primary can update it correctly. In order to get the latest state from the RLDev, the secondary acquires the *map lock* in shared mode, reads the mapping from the RLDev, releases the lock, and never acquires it again. The primary, on the other hand, always acquires the *map lock* in exclusive mode before making mapping changes. The ordinary locking mechanism will then ensure the consistency of the data.

If the primary dies, the secondary will notice it via the failure detector, whereupon it acquires the *map lock* in exclusive mode and acts as a primary. If the secondary

dies, the primary will also detect it via the failure detector. It continues to update the mappings on the RLDev, but does not make RPCs until it gets a revocation for the *map lock* indicating that the secondary has come alive and wants to read the state.

Our design of the chunk manager is very simple, largely because of our decisions to (a) implement replication below the level of the chunk manager, and (b) use the locking service to do the failover.

Figure 3 shows the relationship of the chunk manager and the RLDev layer.

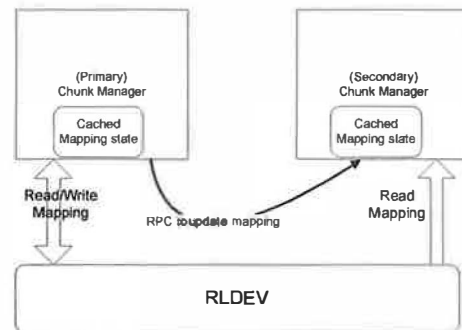


Figure 3: The chunk manager pair relies on a shared RLDev and RPCs to keep the mapping information consistent.

Mapping From Opaque Handles to Disk Offsets

An opaque handle consists of a 32-bit chunk manager identifier and a 64-bit handle identifier. A chunk manager identifier corresponds to a pair of chunk managers. The locations of the primary and secondary chunk manager, which may change over time, are maintained in Paxos and cached by the chunk manager clerk. The clerk uses this mapping to direct chunk requests to an appropriate manager. If the cache is out of date, there is no correctness issue with misdirecting the RPC because the incorrect chunk manager will return an error, which causes the clerk to refresh its mappings.

The translation of the handle identifier to the RLDev is performed by the chunk manager responsible for the handle. It consults its cached mapping table to translate to the RLDev offset. It then invokes the RLDev clerk, which accesses the physical disks as necessary. In most cases, at least one segment of the RLDev will be on a local disk, so accesses incur little network overhead.

The chunk manager module guarantees that an opaque handle is never reused. It enforces this condition by never reusing a handle identifier once it is deallocated. The 64-bit identifier in our current implementation seems adequate for this purpose; but we may reconsider that decision after we gain more experience with the system.

The mapping between handles and disk offsets is stored as stable state on an RLDev. The state on the RLDev consists of a checkpoint and a separate list of incremental changes that have not been applied to the checkpoint. The list is updated synchronously whenever the mapping changes, but the checkpoint is only accessed infrequently. We periodically apply the changes in the list to the checkpoint and truncate the list.

Recovering the chunk manager is simple as long as the RLDev holding the stable state is available. The primary chunk manager applies the list of incremental changes to the current checkpoint to get the updated state, which it caches in memory and stores stably on disk. Then it truncates the list and is ready to service new requests. Since the chunk manager depends on the RLDev for its recovery, the RLDevs have to be recovered first.

3.6 Transaction and Logging Service

Boxwood provides simple transaction and logging support to clients. We provide logging to perform both *redo* and *undo* operations of transactions. Logs are duplexed by storing them on an RLDev so that they are resilient to single failures and universally accessible from any machine. Thus, recovery for a service can be done on any machine. The logging system supports group commits and also allows clients the option of selectively flushing the volatile in-core log to disk on every transaction commit.

Our transaction system does not provide isolation guarantees; instead clients explicitly take out locks using the lock server. Clients do deadlock avoidance by using lock ordering. With the current set of clients in our system, deadlock avoidance is the more attractive alternative to providing deadlock detection as in a traditional transaction system.

3.7 The B-tree Module

We assume the reader is familiar with the B-tree [3], and its variant the *B*-tree* [33], which has the same structure as a B-tree, except that all genuine keys reside in the leaf nodes. Non-leaf nodes contain shadow keys acting purely as an index for finding the desired key in a leaf. A *B-link tree* [22] is a B*-tree with one extra pointer per node: this extra link points to the next node of the tree at the same level as the current node. The extra links make it trivial to enumerate keys in a B-link tree (one just follows the extra links from one leaf node to the next). But even more importantly, the extra links make efficient concurrent operations possible. We do not give details here, but the main intuition is that certain operations can recover from unexpected situations by following the extra links, so some concurrent operations that

would otherwise require several locks can proceed with fewer locks.

The B-link tree, together with algorithms for efficient concurrent operations, was first introduced by Lehman and Yao [22], and significantly improved by Sagiv [28]. Sagiv's algorithms require no locks for lookups. Insertions require only a single lock held at a time even if the insertions cause node splits at many levels of the tree. Each lock protects a single tree node that is being updated. Deletions are handled like insertions, and hold a single lock on the updated node. Thus, these operations are provably deadlock free. Deletions can leave a node empty or partially empty; and such nodes are fixed up as a separate activity by one or more background compression threads. Each compression thread must lock up to three nodes simultaneously. Each B-link tree operation provides all ACID properties when only a single tree is involved. ACID properties across multiple trees must be maintained by clients using the transaction and locking services described earlier.

We implement a distributed version of Sagiv's algorithm in a conceptually simple fashion by using the global lock server instead of simple locks, and by using the chunk manager for storage instead of ordinary disks. Each instance of the B-tree module maintains a write-ahead log on an RLDev.

Recovering the B-tree service is done by replaying the write-ahead log. The records in the log refer to handles implemented by the chunk manager. Thus, before the B-tree service can be recovered, both the RLDev service and the chunk manager must be available. If a B-tree server were to crash, the active log records (i.e., records for those operations that haven't made it to disk) will be protected by leases. When the leases expire, the lock service will initiate recovery using some other B-tree server and the write-ahead log. (Recall from Section 3.3 that clients of the lock service can designate a recovery function to be invoked on lease expiration, when the lock service will consider the client as having failed.) When recovery is complete, the expired leases are made available for other servers to acquire. It is possible that all B-tree servers are dead, in which case, the lock server will not be able to call recovery on any machine when the leases expire. It therefore defers the recovery until the first B-tree server registers with it, at which point all server logs will be replayed before any new leases are handed out.

4 Performance of the Boxwood Prototype

4.1 Experimental Setup

Boxwood is deployed in our lab on a cluster of eight machines connected by a Gigabit Ethernet switch. Each machine is a Dell PowerEdge 2650 server with a single

2.4 GHz Xeon processor, 1GB of RAM, with an Adaptec AIC-7899 dual SCSI adapter, and 5 SCSI drives. One of these, a 36GB 15K RPM (Maxtor Atlas15K) drive, is used as the system disk. The remaining four 18GB 15K RPM drives (Seagate Cheetah 15K.3 ST318453LC) store data.

Each machine executes the Boxwood software, which runs as a user-level process on a Windows Server 2003 kernel. The failure detector observers, the Paxos service, and the lock service run as separate processes; the rest of the layers are available as libraries that can be linked in by applications. Boxwood software is written in C#, a garbage-collected, strongly typed language, with a few low-level routines written in C.

The networking subsystem provided by the kernel is capable of transmitting data at 115 MB/sec using TCP. Using a request response protocol layered on TCP, our RPC system can deliver about 110 MB/sec.

4.2 RLDev Performance

On each machine we run an RLDev server, which manages several RLDevs on its locally attached disks. For some of these RLDevs, the server is a primary and for others it is a secondary. We measure the performance using a simple benchmark program that performs read and write accesses.

Size bytes	Write			Read		
	Xput MB/s	Util. %		Xput MB/s	Util. %	
		CPU	Disk		CPU	Disk
512	0.3	2	99	0.3	20	100
8192	5	9	99	5.3	15	99
64K	40	73	98	48	70	85
256K	110	52	76	110	70	60

Table 1: **Aggregate random write and read throughput on a two machine configuration.** Disk queue length is constrained to be one. Each write request involves one local disk write and one remote disk write. Each read request involves one RPC call and a disk read on a remote disk.

Table 1 shows the performance of replicated random writes and reads in the smallest (two machine) configuration. All the write accesses made by the benchmark are to RLDevs where the primary server is local. Thus, each write will result in a local disk write and an RPC to a remote server to update the mirror. All read accesses are made to a remote RLDev server. CPU utilization represents the average CPU usage on each server. We constrain the kernel disk access routines to allow only a single outstanding request per disk. This represents the most pessimistic performance.

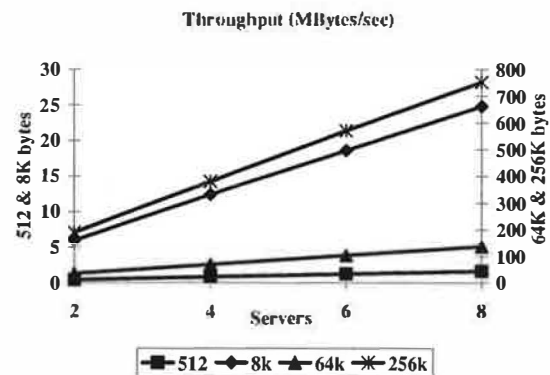


Figure 4: Scaling of writes in the RLDev.

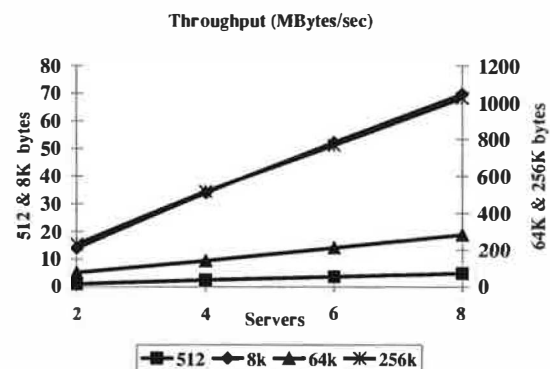


Figure 5: Scaling of reads in the RLDev.

Figures 4 and 5 show the scaling as the number of servers is increased from 2 to 8. Read are slightly faster than writes in general. This is because the disk inherently has slightly better (about 5%) read performance than write according to the manufacturer's specification sheet. At small packet sizes, the throughput is limited by disk latency. For large packet sizes, we get performance close to the RPC system imposed limit. In all cases, we observe good scaling.

4.3 Chunk Manager Performance

We next discuss the performance of the chunk manager. The read and write performance of the chunk manager closely matches the performance of the RLDev described above and is not repeated here. We instead describe the performance impact of allocations and deallocations. Our chunk manager implements a performance optimization that defers some of the work of an allocation to a later time. When possible, we log the allocation locally on the client, but avoid an RPC to the server. A whole set of allocation requests are subsequently sent in a batch to the server.

Table 2 shows the performance effect as the amount of batching is varied. As part of allocating a handle, the

Batch Size	Amortized Latency (ms)
1	24
10	3.3
100	1.0
1000	1.0

Table 2: **Effect of batching on allocations.** Each allocation is for a single 8KB region, which is zeroed out on disk. Latency is amortized over the batch size.

storage allocated for it is zeroed out on disk. Thus allocation costs will vary somewhat with chunk size; we show a typical size of 8KB. For single allocations (i.e., with no batching), the latency cost is high. This is because we require three RLDev writes to stably record the new mapping. However, batching is very effective even at small numbers; we typically more than a dozen allocations that can be batched in our tests.

Deallocation is somewhat easier than allocations. The cost of a single deallocation as seen by a client is about 5.3 milliseconds and is independent of the number of servers. In addition, if a handle whose allocation request has not yet been sent to the server is deallocated, we can avoid making both operations at the server. This brings the time down even further.

4.4 B-tree Performance

We report on the performance of two sets of experiments for the B-tree. In the first experiment we have a number of machines each inserting keys into a separate B-tree. In a subsequent phase, each machine looks up these keys and in a final phase deletes them. Before each of the three phases, all entries in the B-tree caches are evicted. Since the B-trees are private, there is no lock contention, but there is contention for storage on the underlying RLDevs, which are evenly distributed across all machines.

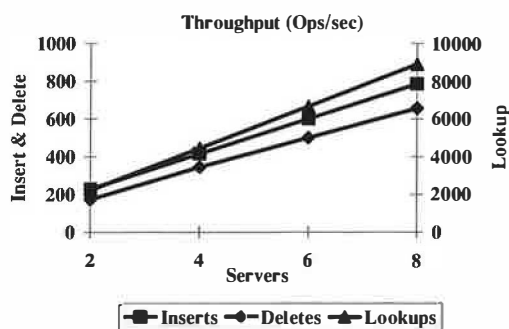


Figure 6: **Performance of B-link tree operations.** Each machine performs operations on a separate tree, starting with a cold cache.

Figure 6 reports on the aggregate steady state throughput across all machines. For all phases, we observe good scaling within experimental error. This is not surprising since there is no contention for the locks, and the RLDev performance is known to scale from Figures 4 and 5.

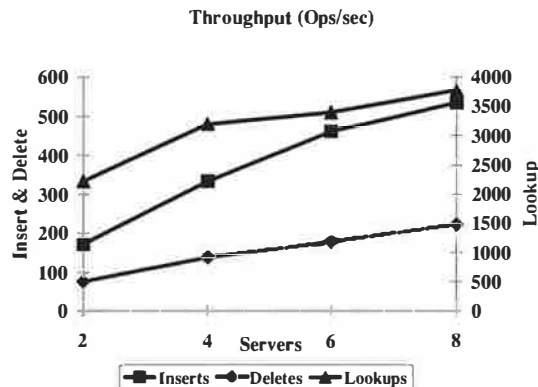


Figure 7: **Performance of B-link tree operations.** All machines perform operations on a single shared tree, starting with a cold cache.

We next study the performance of B-tree operations under contention. In this experiment, we redo the previous experiment but use a single B-tree that all machines contend for. To avoid the extreme case where every insertion contends for a global lock, we partition the keys so that each client acts on a disjoint region of the key space. This arrangement still leads to lock contention in the upper levels of the tree. Once again we perform three phases with cold caches. Figure 7 shows the aggregate throughput across all machines. Under contention, the performance of the B-link operations tends to flatten relative to Figure 6. Also, relative to the first experiment, the size the shared tree is proportionately larger in the second experiment. This leads to lowered hit rates in the cache, which slows the performance of lookups in the second case.

In general, the performance of B-trees is dependent on several parameters, which include the branching factor, the distribution of keys, and the size of the keys and data. We have not yet done an exhaustive characterization of our B-tree module at various parameter values. The particular trees we measure have 10-byte keys and data and each tree node has a branching factor of 6.

5 BoxFS: A Multi-Node NFS Server

The previous sections showed the functional characteristics of Boxwood's abstractions. To test these abstractions, we built BoxFS: a file system using Boxwood B-trees, exported using the NFS v2 protocol. It runs at user-level and directly or indirectly depends on all the services

described earlier.

BoxFS implements a conventional file system containing directories, files, and symbolic links. Each of these is stored as a Boxwood B-tree. Each tree contains a single well-known *distinguished key* to hold attributes such as access times, protection and owner information similar to a UNIX *inode*. In addition, directories, files, and symbolic links also have *regular keys* as described below.

A B-tree storing a directory is keyed by the name of the file, directory, or symbolic link. The data corresponding to the key refers to a byte array that is the on-wire representation of the NFS file handle. The NFS file handle is not interpreted by the client, but BoxFS stores in it bookkeeping information including the B-Tree handle that represents the file, directory, or symbolic link.

A B-tree storing a file is keyed by block number. The data corresponding to the key is an opaque chunk handle. The actual data in the file is stored in the chunk as uninterpreted bytes. Our usage of B-trees for directories and chunks for file data is consistent with the idea that strict atomicity guarantees are needed in the file system metadata, but not in the file user data. Alternatively, we could have stored user data as part of the file B-tree, but every file write would translate into a relatively expensive B-tree insert operation, with attendant overheads of redo and undo logging.

A B-tree storing a symbolic link is a degenerate tree containing a single special key and the byte array referring to the target of the link.

Obviously, all the B-trees are maintained via Boxwood B-tree operations — and any single such operation has all the ACID properties even when the same B-tree is simultaneously being accessed by different machines. However, many file system operations require multiple B-tree operations to be performed atomically. For example, a *rename* operation requires the combination of a delete from one B-tree and an insertion into another B-tree to be atomic. This atomicity is achieved using the transaction service described in Section 3.6. But as we explained earlier, clients have to provide isolation guarantees themselves. BoxFS ensures isolation by acquiring locks on the appropriate file system objects from the Boxwood lock service in a predefined lock order. After the locks are acquired, it performs the required operation as part of a transaction, commits, and releases the locks.

All B-tree operations benefit from the B-tree cache built into Boxwood's B-tree module. In contrast, file data is not stored in B-trees; it is accessed via the chunk manager interface and requires a separate cache within BoxFS. This cache is kept coherent by acquiring the appropriate shared or exclusive lock from the Boxwood lock service for every data read or write.

Typical file operations that do not involve user file data access result in one or more B-tree operations encapsu-

lated within a single transaction. These B-tree operations will generate log entries that are committed to the log when the transaction commits. Where possible the transaction system will do a group commit. In any event, a single transaction commit will cause only a single disk write in the usual case unless the log is full. The actual B-tree nodes that are modified by the transaction will remain in the B-tree's in-core cache and do not make it to disk. Thus, BoxFS can perform metadata intensive file operations efficiently.

BoxFS makes three simplifying assumptions to achieve acceptable performance. We do not believe these assumptions materially affect the usability of our system or the validity of our hypothesis that it is easy to build file systems given higher-level abstractions. By default, the data cache is flushed once every 30 seconds, which is not strictly in accordance with NFS v2 semantics. Also, the B-tree log that contains the metadata operations is flushed to stable storage with the same periodicity. Thus, file system metadata is consistent, but we could lose 30 seconds of work if a machine crashes. Finally, we do not always keep the access times on files and directories up to date.

The BoxFS system runs on multiple machines simultaneously and exports the same file system. The file system is kept consistent by virtue of the distributed lock service. However, since we are exporting the file system using the NFS protocol, clients cannot fully exploit the benefits of coherent caching.

Locking in BoxFS is quite fine-grained. Since all file system metadata is stored as key-data pairs in a B-tree, metadata modifications made by different machines are automatically locked for consistency by the B-tree module through the global lock server. In addition, BoxFS protects individual file blocks by taking out locks for each block. Thus, two machines writing different blocks in the same file will contend for the metadata lock to update the file attributes, but not for the individual block. This reduces lock contention as well as cache coherence traffic due to false sharing.

Using the Boxwood abstractions enabled us to keep our file system code base small. The actual file system code is about 1700 lines of C# code, which is a more verbose language than C. The BoxFS code implementing a simple LRU buffer cache is an additional 800 lines, which is largely a reuse of the code in the B-tree cache module. The size of the code compares favorably with Mark Shand's classic user-level NFS daemon, which is about 2500 lines of C for read-only access to a UNIX file hierarchy [30]. In addition, we wrote code to support NFS RPC (both UDP and TCP) and XDR in C#, which accounted for an additional 2000 lines, and about 1000 lines of C to interface BoxFS to the native Windows networking libraries.

5.1 BoxFS Performance

Table 3 shows Connectathon performance benchmarks (available at <http://www.connectathon.org/nfstests.html>) for BoxFS. For comparison we show the performance of a stock NFS server running on the same hardware. We run BoxFS on a single machine in our cluster. The RLDevs are on four locally attached disks and have no replication enabled. The stock NFS server is the Windows 2003 Services for Unix (SFU) NFS server that runs in-kernel on the NTFS local file system on the same four local disks. Our client is a Linux 2.4.20-8 NFS client in both cases. We mount the file system so that NFS RPCs are made with UDP send and receive sizes of 8KB, the maximum supported by the kernel.

Description	Time (secs)	
	BoxFS	NFS
Create 155 files and 62 dirs	0.7(1.9)	1.0
Remove 155 files and 62 dirs.	0.5(1.7)	0.4
500 Getwd and stat calls	0.1(0.1)	0.8
1000 Chmods and stats	1.5(4.8)	8.4
Write a 1MB file 10 times	3.7(12.4)	10.8
Read a 1MB file 10 times	0.2(0.2)	0.2
Reads 20500 dir. entries and 200 files and unlinks them	1.5(2.7)	0.6
200 Renames and links on 10 files	0.9(2.3)	1.9
400 Symlinks and readlinks on 100 files	1.4(3.7)	6.7
1500 Statfs calls	1.2(1.3)	1.1

Table 3: **Connectathon benchmarks.** BoxFS writes file data asynchronously. The test that reads a 1MB file repeatedly is an anomaly because the reads hit in the cache on the client. The numbers in parenthesis indicate performance when the in-core metadata log is flushed to disk on each transaction commit instead of periodically.

We show the performance of BoxFS when the metadata log is lazily flushed to disk and also when it is not. In general, the performance of BoxFS is comparable to the native NFS server on top of a local NTFS file system. We have a slight edge on some meta-data intensive operations, which we attribute to our usage of B-trees. Depending on whether we synchronously update the log or not, the fifth test shows a marked difference (12.4 versus 3.7 seconds) even though file data is being updated asynchronously in both cases. This is because file write requires a metadata update, which involves a log write.

To show the benefit of lazily updating the metadata log where possible, we report our results from running PostMark, a metadata intensive benchmark. This benchmark models the expected file system work load imposed by electronic mail, web based commerce, and netnews [18]. It creates an initial set of files of varying sizes. This

Parameter	Value
Num. Initial Files	500
Num. of Transactions	500
File Sizes	0.5–9.8KB
Create/Delete Ratio	5
Read/Append Ratio	5
Read/Write Block Size	512

Table 4: **Postmark parameters.** We use Unix buffered I/O and use a default random number seed of 42.

Metric	BoxFS	
	Async.	Sync.
File Creations/sec	63	27
File Read/sec	81	24
File Append/sec	85	25
File Deletions/sec	63	27
Data Read KB/sec	116	50
Data Write KB/sec	379	162

Table 5: **Postmark Results.** We gain substantial performance by flushing the meta data log to disk periodically, without significant change in the semantics of the file system.

measures the performance of the file system in creating small files. Next it measures the performance of a predetermined number of “transactions”, where a transaction is either the creation or deletion of a file followed by either a read or append to a file. We ran the benchmark with the settings shown in Table 4. Our results are shown in Table 5. Since the benchmark emphasizes metadata intensive operations, BoxFS performs well.

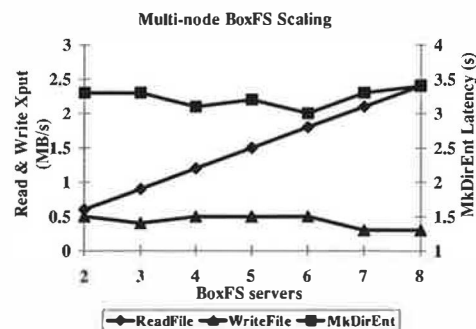


Figure 8: **Performance of BoxFS sharing experiments.** MKDirEnt Latency is the elapsed time to insert all 100 files.

Since benchmarks for multi-node file systems like BoxFS are difficult to obtain, we show scaling characteristics of BoxFS on three simple experiments in Figure 8. In all three experiments, we run the Boxwood abstractions on all 8 machines in our cluster. We export a single shared NFS volume from one or more of these machines.

By the nature of NFS mounting, the single volume appears as separate file systems to the client.

In the first experiment (*ReadFile* in Figure 8), a single 1 MB NFS file is read from multiple mount points, with a cold client cache and a cold server cache. As we would expect, the aggregate throughput increases linearly as the number of servers is increased since there is no contention.

In the second experiment (*MkDirEnt* in the figure), we create 100 files with unique names in the root directory of each NFS volume. This results in NFS Create RPC requests to each BoxFS machine, which in turn modifies the B-tree corresponding to the single shared directory. There are two potential sources of conflict traffic. First, we are inserting into a shared B-tree, which will result in locking and data transfer in the B-tree module; second, BoxFS has to acquire a lock to ensure that the metadata update and the B-tree insertion is consistently done. Since our experiments in Figure 7 show that we can perform in excess of 300 insertions a second into the B-tree, we believe the performance bottleneck is the traffic to keep the metadata up to date. For comparison, the performance when there is no sharing (single machine case) is 0.6 ms.

In the final experiment (*WriteFile* in the figure), we write data to a shared file, but at non-overlapping offsets. Each write results in an NFS Write RPC to a different BoxFS machine. Since BoxFS implements fine-grained locking at the file block level, the coherence traffic and lock contention, which determines scaling, is limited to what is required to keep the file metadata up to date. For comparison, the performance in the single machine (no contention) case is 4 MB/sec.

6 Related Work

Some early seeds of our work are present in the “scalable distributed data structures” (SDDSs) of Litwin *et al.* [24, 25, 26]. Litwin’s SDDSs offer algorithms for interacting in a scalable fashion with a particular family of data structures. Our focus in Boxwood is significantly different, in that we are concerned with systems issues such as reliability and fault-tolerance in general abstractions—issues that were largely ignored in Litwin’s work.

The “scalable distributed data structure” approach of Gribble *et al.* [13] is the previous work most similar to Boxwood. We view our work as complementary to theirs. The implementation of different data structures (hash tables versus B-trees) offers very different tradeoffs. Furthermore, our failure models are different, which also lead to very different system design techniques. Gribble *et al.* designed their system assuming the availability of uninterruptible power supplies, so tran-

sient failures that take down the entire cluster are extremely rare. They rely on data always being available in more than one place (in RAM or on disk on multiple machines) to design their recovery protocols.

Production systems like BerkeleyDB [15] and DataBlade are related to our approach and were sources of early inspiration for our work. These systems offer much more functionality than we do at the moment, but neither is distributed.

Petal [21] is another related approach to scalable storage. Petal provides similar durability and consistency guarantees to Boxwood, but it presents applications with a single sparse address space of blocks. Applications must coherently manage this space themselves, and implement their own logging and recovery mechanisms for any data structures they employ. Many of the basic services that Petal uses are similar in spirit to ours. Similarly, the overall structure of the B-link tree service is reminiscent of Frangipani [32].

Distributed databases are another approach to scalably storing information (e.g., [23]). However, we view our system as lower-level infrastructure that we hope database designers will use (and we plan to use it in this way ourselves). We have deliberately avoided mechanisms that we felt are unnecessary in the lower levels of storage architecture: query parsing, deadlock detection, and full-fledged transactions.

Distributed file systems also enable storage to scale in some respects (e.g., [2, 10] to name but two). But, once again, our view is that these will be layered over the facilities of Boxwood.

Boxwood’s layering of a file system on top of B-trees is related to Olson’s Inversion File System, which is layered on top of a full-fledged Postgres database [27]. The Inversion File System is capable of answering queries about the file system that our implementation cannot. Since our file system is layered on a simpler abstraction, all things being equal, we expect a performance increase at the expense of reduced querying capability.

Recent work on Semantically Smart Disks [31] is also related to Boxwood. Like Boxwood, semantically smart disks strive for better file system performance by exploiting usage patterns and other semantics within the lower layers of the storage system. Unlike Boxwood, semantically smart disks present a conventional disk interface (e.g., SCSI) to higher levels, but try to encapsulate (or infer) semantic knowledge about the file system to enhance performance.

Many projects, including FARSITE [1], OceanStore [19], and the rapidly-growing literature on distributed hash tables (e.g., [34]), have attempted to provide scalable storage over a wide area network. These all address a set of trade-offs quite different to those of Boxwood. The wide-area solutions must deal

with untrusted participants, frequent reconfiguration, high network latency, and variable bandwidth. Thus, these solutions do not take advantage of the more benign conditions for which Boxwood was designed.

There are also many projects using local area clusters to provide scalable services or persistent distributed objects. Typically, these rely on a file system or database to manage their storage (e.g., [7, 8]) or build an application-specific storage service (e.g., [29]) and do not offer easily-utilized, atomically-updating data structures.

Network attached secure disks (NASD) [11] are also related to Boxwood in that they provide a storage abstraction at a higher level than raw blocks.

Our choice of the B-link tree was motivated by the observation that it is well suited for distributed implementation. A similar observation was made independently by Johnson and Colbrook [16], who proposed B-link tree variants called DE- and DB-trees for shared memory multiprocessors. Their scheme explicitly replicates internal nodes on specific processors and stores contiguous sets of keys on a processor. Our implementation is more dynamic and our algorithm is simpler at the cost of having a distributed lock service.

7 Conclusions

Our initial experience indicates that using scalable data abstractions as fundamental, low-level storage primitives is attractive. To be sure, it appears difficult to settle on a single, universal abstraction that will fit all needs. However, our particular combination of abstractions and services seem to offer a sound substrate, on top of which multiple abstractions may be readily built.

Our use of the chunk manager as a generalized storage allocator obviating the need for address space management elsewhere in the system seems to be widely applicable. In our case, it enabled us to distribute a complicated data structure with modest programming effort.

Our strategy of isolating the use of the Paxos module to a relatively small part of the system has worked well in practice. It allows us to continue to scale the rest of the system dynamically without much hindrance.

Does the design of BoxFS support our claim that building scalable applications using the Boxwood infrastructure is easy? This is a subjective question, but our feeling on this is positive. BoxFS is a distributed file system with good resilience and scalability. Yet it does not require complicated locking, logging, or recovery schemes, making it very easy to implement. We need further performance analysis to make more objective comparisons.

Acknowledgements

We wish to thank current and former colleagues Andrew Birrell, Mike Burrows, Úlfar Erlingsson, Jim Gray, Ed Lee, Roy Levin, and Mike Schroeder for many interesting discussions related to the ideas in this paper. Ahmed Talat, Roopesh Battepati, and Ahmed Mohamed helped us understand Windows networking and SFU/NFS. Qin Lv and Feng Zhou, who were interns during two successive summers, helped us understand several subtle issues with our design. We also thank our anonymous reviewers and our shepherd, Jason Flinn, for their comments on the paper.

References

- [1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. 5th Symp. Operating Systems Design and Implementation (OSDI)*, pages 1–14, December 2002.
- [2] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. In *Proc. 15th Symp. Operating Systems Principles (SOSP)*, pages 109–126, December 1995.
- [3] R. Bayer and C. McCreight. Organization and maintenance of large ordered indexes. *Acta Inf.*, 1(3):173–189, 1972.
- [4] W. M. Cardoza, F. S. Glover, and W. E. Snaman, Jr. Design of the TruCluster multicomputer system for the Digital UNIX environment. *Digital Technical Journal*, 8(1):5–17, 1996.
- [5] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM (JACM)*, 43(4):685–722, July 1996.
- [6] M. Devarakonda, B. Kish, and A. Mohindra. Recovery in the Calypso file system. *ACM Transactions on Computer Systems*, 14(3):287–310, August 1996.
- [7] P. Ferreira, M. Shapiro, X. Blondel, O. Fambon, J. Garcia, S. Kloosterman, N. Richer, M. Robert, F. Sandalky, G. Coulouris, J. Dollimore, P. Guedes, D. Hagimont, and S. Krakowiak. PerDiS: Design, implementation, and use of a persistent distributed store. In *Recent Advances in Distributed Systems*, volume 1752 of *Lecture Notes in Computer Science*, chapter 18, pages 427–452. Springer-Verlag, February 2000.
- [8] A. Fox, S. Gribble, Y. Chawathe, E. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proc. 16th Symp. Operating Systems Principles (SOSP)*, pages 78–91, October 1997.
- [9] S. Frølund, A. Merchant, Y. Saito, S. Spence, and A. Veitch. FAB: Building reliable enterprise storage systems on the cheap. In *Proc. 11th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2004.

- [10] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *Proc. 19th Symp. Operating Systems Principles (SOSP)*, pages 29–43, December 2003.
- [11] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proc. 8th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 92–103, October 1998.
- [12] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proc. 12th Symp. Operating Systems Principles (SOSP)*, pages 202–210, December 1989.
- [13] S. Gribble, E. Brewer, J. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In *Proc. 4th Symp. on Operating Systems Design and Implementation (OSDI)*, pages 319–332, October 2000.
- [14] H. Hsiao and D. J. DeWitt. Chained declustering: A new availability strategy for multiprocessor database machines. In *Proc. of the 6th International Conference on Data Engineering*, pages 456–465, February 1990.
- [15] SleepyCat Software Inc. *Berkeley DB*. New Riders Publishing, 2002.
- [16] T. Johnson and A. Colbrook. A distributed data-balanced dictionary based on the B-link tree. Technical Report MIT/LCS/TR-530, MIT, 1992.
- [17] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The logical disk: A new approach to improving file systems. In *Proc. 14th Symp. Operating Systems Principles (SOSP)*, pages 15–28, December 1993.
- [18] J. Katcher. PostMark: A new file system benchmark. Technical Report TR3022, NetApp, 2004.
- [19] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weather-
spoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proc. 9th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 190–201, November 2000.
- [20] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.
- [21] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proc. 7th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 84–92, October 1996.
- [22] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Systems*, 6(4):650–670, 1981.
- [23] B. Lindsay. A retrospective of R*. *Proc. IEEE*, 75(8):668–673, 1987.
- [24] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proc. 6th Int. Conf. Very Large Data Bases (VLDB)*, pages 212–223, October 1980.
- [25] W. Litwin, M.-A. Neimat, and D. Schneider. RP*: A family of order preserving scalable distributed data structures. In *Proc. 20th Int. Conf. Very Large Data Bases (VLDB)*, pages 342–353, September 1994.
- [26] W. Litwin, M.-A. Neimat, and D. A. Schneider. LH* — a scalable, distributed data structure. *ACM Trans. Database Systems*, 21(4):480–525, 1996.
- [27] M. A. Olson. The design and implementation of the Inversion file system. In *Proc. of the USENIX Winter 1993 Technical Conference*, pages 205–217, January 1993.
- [28] Y. Sagiv. Concurrent operations on B*-trees with overtaking. *Journal Computer and System Sciences*, 33:275–296, 1986.
- [29] Y. Saito, B. Bershad, and H. Levy. Manageability, availability and performance in Porcupine: A highly scalable, cluster-based mail service. In *Proc. 17th Symp. Operating Systems Principles (SOSP)*, pages 1–15, December 1999.
- [30] M. A. Shand. UNFSD—User-level NFS server. *comp.sources.unix*, 15(No. 1 and 2), May 1988.
- [31] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-smart disk systems. In *2nd USENIX Conference on File and Storage Technologies (FAST)*, pages 73–88, March 2003.
- [32] C. A. Thekkath, T. P. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proc. 16th Symp. Operating Systems Principles (SOSP)*, pages 224–237, October 1997.
- [33] H. Wedekind. On the selection of access paths in a database system. In J. W. Klimbie and K. L. Koffeman, editors, *Data Base Management*, pages 385–397. North-Holland, Amsterdam, 1974.
- [34] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal Selected Areas in Communications (JSAC)*, 22(1), 2004.

Secure Untrusted Data Repository (SUNDR)

Jinyuan Li, Maxwell Krohn*, David Mazières, and Dennis Shasha
NYU Department of Computer Science

Abstract

SUNDR is a network file system designed to store data securely on untrusted servers. SUNDR lets clients detect any attempts at unauthorized file modification by malicious server operators or users. SUNDR's protocol achieves a property called *fork consistency*, which guarantees that clients can detect any integrity or consistency failures as long as they see each other's file modifications. An implementation is described that performs comparably with NFS (sometimes better and sometimes worse), while offering significantly stronger security.

1 Introduction

SUNDR is a network file system that addresses a long-standing tension between data integrity and accessibility. Protecting data is often viewed as the problem of building a better fence around storage servers—limiting the number of people with access, disabling unnecessary software that might be remotely exploitable, and staying current with security patches. This approach has two drawbacks. First, experience shows that people frequently do not build high enough fences (or sometimes entrust fences to administrators who are not completely trustworthy). Second and more important, high fences are inconvenient; they restrict the ways in which people can access, update, and manage data.

This tension is particularly evident for free software source code repositories. Free software projects often involve geographically dispersed developers committing source changes from all around the Internet, making it impractical to fend off attackers with firewalls. Hosting code repositories also requires a palette of tools such as CVS [4] and SSH [35], many of which have had remotely exploitable bugs.

Worse yet, many projects rely on third-party hosting services that centralize responsibility for large numbers of otherwise independent code repositories. `sourceforge.net`, for example, hosts CVS repositories

for over 20,000 different software packages. Many of these packages are bundled with various operating system distributions, often without a meaningful audit. By compromising `sourceforge`, an attacker can therefore introduce subtle vulnerabilities in software that may eventually run on thousands or even millions of machines.

Such concerns are no mere academic exercise. For example, the Debian GNU/Linux development cluster was compromised in 2003 [2]. An unauthorized attacker used a sniffed password and a kernel vulnerability to gain superuser access to Debian's primary CVS and Web servers. After detecting the break-in, administrators were forced to freeze development for several days, as they employed manual and ad-hoc sanity checks to assess the extent of the damage. Similar attacks have also succeeded against Apache [1], Gnome [32], and other popular projects.

Rather than hope for invulnerable servers, we have developed SUNDR, a network file system that reduces the need to trust storage servers in the first place. SUNDR cryptographically protects all file system contents so that clients can detect any unauthorized attempts to change files. In contrast to previous Byzantine-fault-tolerant file systems [6, 27] that distribute trust but assume a threshold fraction of honest servers, SUNDR vests the authority to write files entirely in users' public keys. Even a malicious user who gains complete administrative control of a SUNDR server cannot convince clients to accept altered contents of files he lacks permission to write.

Because of its security properties, SUNDR also creates new options for managing data. By using SUNDR, organizations can outsource storage management without fear of server operators tampering with data. SUNDR also enables new options for data backup and recovery: after a disaster, a SUNDR server can recover file system data from untrusted clients' file caches. Since clients always cryptographically verify the file system's state, they are indifferent to whether data was recovered from untrusted clients or resided on the untrusted server all along.

This paper details the SUNDR file system's design and implementation. We first describe SUNDR's security protocol and then present a prototype implementation that gives performance generally comparable to the popular

*now at MIT CS & AI Lab

NFS file system under both an example software development workload and microbenchmarks. Our results show that applications like CVS can benefit from SUNDR's strong security guarantees while paying a digestible performance penalty.

2 Setting

SUNDR provides a file system interface to remote storage, like NFS [29] and other network file systems. To secure a source code repository, for instance, members of a project can mount a remote SUNDR file system on directory `/sundr` and use `/sundr/cvsroot` as a CVS repository. All checkouts and commits then take place through SUNDR, ensuring users will detect any attempts by the hosting site to tamper with repository contents.

Figure 1 shows SUNDR's basic architecture. When applications access the file system, the client software internally translates their system calls into a series of *fetch* and *modify* operations, where *fetch* means retrieving a file's contents or validating a cached local copy, and *modify* means making new file system state visible to other users. Fetch and modify, in turn, are implemented in terms of SUNDR protocol RPCs to the server. Section 3 explains the protocol, while Section 5 describes the server design.

To set up a SUNDR server, one runs the server software on a networked machine with dedicated SUNDR disks or partitions. The server can then host one or more file systems. To create a file system, one generates a public/private *superuser* signature key pair and gives the public key to the server, while keeping the private key secret. The private key provides exclusive write access to the root directory of the file system. It also directly or indirectly allows access to any file below the root. However, the privileges are confined to that one file system. Thus, when a SUNDR server hosts multiple file systems with different superusers, no single person has write access to all files.

Each user of a SUNDR file system also has a signature key. When establishing an account, users exchange public keys with the superuser. The superuser manages accounts with two superuser-owned file in the root directory of the file system: `.sundr.users` lists users' public keys and numeric IDs, while `.sundr.group` designates groups and their membership. To mount a file system, one must specify the superuser's public key as a command-line argument to the client, and must furthermore give the client access to a private key. (SUNDR could equally well manage keys and groups with more flexible certificate schemes; the system only requires some way for users to validate each other's keys and group membership.)

Throughout this paper, we use the term *user* to design-

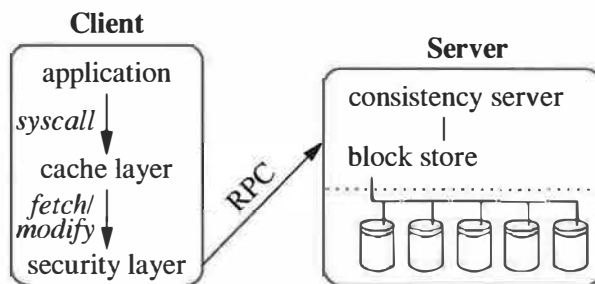


Figure 1: Basic SUNDR architecture.

nate an entity possessing the private half of a signature key mapped to some user ID in the `.sundr.users` file. Depending on context, this can either be the person who owns the private key, or a client using the key to act on behalf of the user. However, SUNDR assumes a user is aware of the last operation he or she has performed. In the implementation, the client remembers the last operation it has performed on behalf of each user. To move between clients, a user needs both his or her private key and the last operation performed on his or her behalf (concisely specified by a version number). Alternatively, one person can employ multiple user IDs (possibly with the same public key) for different clients, assigning all file permissions to a personal group.

SUNDR's architecture draws an important distinction between the administration of servers and the administration of file systems. To administer a server, one does not need any private superuser keys.¹ In fact, for best security, key pairs should be generated on separate, trusted machines, and private keys should never reside on the server, even in memory. Important keys, such as the superuser key, should be stored off line when not in use (for example on a floppy disk, encrypted with a passphrase).

3 The SUNDR protocol

SUNDR's protocol lets clients detect unauthorized attempts to modify files, even by attackers in control of the server. When the server behaves correctly, a *fetch* reflects exactly the authorized modifications that happened before it.² We call this property *fetch-modify consistency*.

If the server is dishonest, clients enforce a slightly

¹The server does actually have its own public key, but only to prevent network attackers from "framing" honest servers; the server key is irrelevant to SUNDR's security against compromised servers.

²Formally, *happens before* can be any irreflexive partial order that preserves the temporal order of non-concurrent operations (as in Linearizability [11]), orders any two operations by the same client, and orders a modification with respect to any other operation on the same file.

weaker property called *fork consistency*. Intuitively, under fork consistency, a dishonest server could cause a fetch by a user *A* to miss a modify by *B*. However, either user will detect the attack upon seeing a subsequent operation by the other. Thus, to perpetuate the deception, the server must fork the two user's views of the file system. Put equivalently, if *A*'s client accepts some modification by *B*, then at least until *B* performed that modification, both users had identical, fetch-modify-consistent views of the file system.

We have formally specified fork consistency [16], and, assuming digital signatures and a collision-resistant hash function, proven SUNDR's protocol achieves it [17]. Therefore, a violation of fork consistency means the underlying cryptography was broken, the implementation deviated from the protocol, or there is a flaw in our mapping from high-level Unix system calls to low-level fetch and modify operations.

In order to discuss the implications of fork consistency and to describe SUNDR, we start with a simple straw-man file system that achieves fork consistency at the cost of great inefficiency (Section 3.1). We then propose an improved system with more reasonable bandwidth requirements called "Serialized SUNDR" (Section 3.3). We finally relax serialization requirements, to arrive at "concurrent SUNDR," the system we have built (Section 3.4).

3.1 A straw-man file system

In the roughest approximation of SUNDR, the straw-man file system, we avoid any concurrent operations and allow the system to consume unreasonable amounts of bandwidth and computation. The server maintains a single, untrusted global lock on the file system. To fetch or modify a file, a user first acquires the lock, then performs the desired operation, then releases the lock. So long as the server is honest, the operations are totally ordered and each operation completes before the next begins.

The straw-man file server stores a complete, ordered list of every fetch or modify operation ever performed. Each operation also contains a digital signature from the user who performed it. The signature covers not just the operation but also *the complete history of all operations that precede it*. For example, after five operations, the history might appear as follows:

fetch(f_2)	mod(f_3)	fetch(f_3)	mod(f_2)	fetch(f_2)
user A	user B	user A	user A	user B
sig	sig	sig	sig	sig

To fetch or modify a file, a client acquires the global lock, downloads the entire history of the file system, and

validates each user's most recent signature. The client also checks that its own user's previous operation is in the downloaded history (unless this is the user's very first operation on the file system).

The client then traverses the operation history to construct a local copy of the file system. For each modify encountered, the client additionally checks that the operation was actually permitted, using the user and group files to validate the signing user against the file's owner or group. If all checks succeed, the client appends a new operation to the list, signs the new history, sends it to the server, and releases the lock. If the operation is a modification, the appended record contains new contents for one or more files or directories.

Now consider, informally, what a malicious server can do. To convince a client of a file modification, the server must send it a signed history. Assuming the server does not know users' keys and cannot forge signatures, any modifications clients accept must actually have been signed by an authorized user. The server can still trick users into signing inappropriate histories, however, by concealing other users' previous operations. For instance, consider what would happen in the last operation of the above history if the server failed to show user *B* the most recent modification to file f_2 . Users *A* and *B* would sign the following histories:

user A:	fetch(f_2)	mod(f_3)	fetch(f_3)	mod(f_2)
	user A	user B	user A	user A
	sig	sig	sig	sig
user B:	fetch(f_2)	mod(f_3)	fetch(f_3)	fetch(f_2)
	user A	user B	user A	user B
	sig	sig	sig	sig

Neither history is a prefix of the other. Since clients always check for their own user's previous operation in the history, from this point on, *A* will sign only extensions of the first history and *B* will sign only extensions of the second. Thus, while before the attack the users enjoyed fetch-modify consistency, after the attack the users have been forked.

Suppose further that the server acts in collusion with malicious users or otherwise comes to possess the signature keys of compromised users. If we restrict the analysis to consider only histories signed by honest (i.e., uncompromised) users, we see that a similar forking property holds. Once two honest users sign incompatible histories, they cannot see each others' subsequent operations without detecting the problem. Of course, since the server can extend and sign compromised users' histories, it can change any files compromised users can write. The re-

maining files, however, can be modified only in honest users' histories and thus continue to be fork consistent.

3.2 Implications of fork consistency

Fork consistency is the strongest notion of integrity possible without on-line trusted parties. Suppose user *A* comes on line, modifies a file, and goes off line. Later, *B* comes on line and reads the file. If *B* doesn't know whether *A* has accessed the file system, it cannot detect an attack in which the server simply discards *A*'s changes. Fork consistency implies this is the only type of undetectable attack by the server on file integrity or consistency. Moreover, if *A* and *B* ever communicate or see each other's future file system operations, they can detect the attack.

Given fork consistency, one can leverage any trusted parties that are on line to gain stronger consistency, even fetch-modify consistency. For instance, as described later in Section 5, the SUNDR server consists of two programs, a block store for handling data, and a consistency server with a very small amount of state. Moving the consistency server to a trusted machine trivially guarantees fetch-modify consistency. The problem is that trusted machines may have worse connectivity or availability than untrusted ones.

To bound the window of inconsistency without placing a trusted machine on the critical path, one can use a "time stamp box" with permission to write a single file. The box could simply update that file through SUNDR every 5 seconds. All users who see the box's updates know they could only have been partitioned from each other in the past 5 seconds. Such boxes could be replicated for Byzantine fault tolerance, each replica updating a single file.

Alternatively, direct client-client communication can be leveraged to increase consistency. Users can write login and logout records with current network addresses to files so as to find each other and continuously exchange information on their latest operations. If a malicious server cannot disrupt network communication between clients, it will be unable to fork the file system state once on-line clients know of each other. Those who deem malicious network partitions serious enough to warrant service delays in the face of client failures can conservatively pause file access during communication outages.

3.3 Serialized SUNDR

The straw-man file system is impractical for two reasons. First, it must record and ship around complete file system operation histories, requiring enormous amounts of bandwidth and storage. Second, the serialization of operations through a global lock is impractical for a multi-user

network file system. This subsection explains SUNDR's solution to the first problem; we describe a simplified file system that still serializes operations with a global lock, but is in other respects similar to SUNDR. Subsection 3.4 explains how SUNDR lets clients execute non-conflicting operations concurrently.

Instead of signing operation histories, as in the straw-man file system, SUNDR effectively takes the approach of signing file system snapshots. Roughly speaking, users sign messages that tie together the complete state of all files with two mechanisms. First, all files writable by a particular user or group are efficiently aggregated into a single hash value called the *i-handle* using *hash trees* [18]. Second, each *i-handle* is tied to the latest version of every other *i-handle* using *version vectors* [23].

3.3.1 Data structures

Before delving into the protocol's details, we begin by describing SUNDR's storage interface and data structures. Like several recent file systems [9, 20], SUNDR names all on-disk data structures by cryptographic handles. The block store indexes most persistent data structures by their 20-byte SHA-1 hashes, making the server a kind of large, high-performance hash table. It is believed to be computationally infeasible to find any two different data blocks with the same SHA-1 hash. Thus, when a client requests the block with a particular hash, it can check the integrity of the response by hashing it. An incidental benefit of hash-based storage is that blocks common to multiple files need be stored only once.

SUNDR also stores messages signed by users. These are indexed by a hash of the public key and an index number (so as to distinguish multiple messages signed by the same key).

Figure 2 shows the persistent data structures SUNDR stores and indexes by hash, as well as the algorithm for computing *i-handles*. Every file is identified by a (principal, *i-number*) pair, where principal is the user or group allowed to write the file, and *i-number* is a per-principal inode number. Directory entries map file names onto (principal, *i-number*) pairs. A per-principal data structure called the *i-table* maps each *i-number* in use to the corresponding inode. User *i-tables* map each *i-number* to a hash of the corresponding inode, which we call the file's *i-hash*. Group *i-tables* add a level of indirection, mapping a group *i-number* onto a user *i-number*. (The indirection allows the same user to perform multiple successive writes to a group-owned file without updating the group's *i-handle*.) Inodes themselves contain SHA-1 hashes of file data blocks and indirect blocks.

Each *i-table* is stored as a B+-tree, where internal nodes

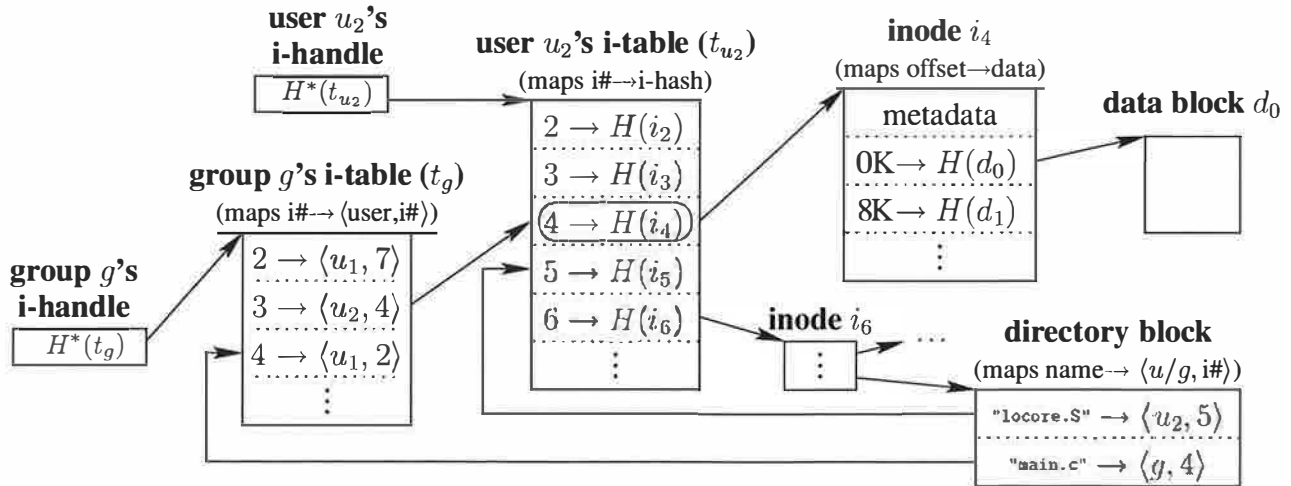


Figure 2: User and group i-handles. An *i-handle* is the root of a hash tree containing a user or group *i-table*. (H denotes SHA-1, while H^* denotes recursive application of SHA-1 to compute the root of a hash tree.) A *group i-table* maps group inode numbers to user inode numbers. A *user i-table* maps a user's inode numbers to i-hashes. An *i-hash* is the hash of an inode, which in turn contains hashes of file data blocks.

contain the SHA-1 hashes of their children, thus forming a hash tree. The hash of the B+-tree root is the i-handle. Since the block store allows blocks to be requested by SHA-1 hash, given a user's i-handle, a client can fetch and verify any block of any file in the user's i-table by recursively requesting the appropriate intermediary blocks. The next question, of course, is how to obtain and verify a user's latest i-handle.

3.3.2 Protocol

i-handles are stored in digitally-signed messages known as *version structures*, shown in Figure 3. Each version structure is signed by a particular user. The structure must always contain the user's i-handle. In addition, it can optionally contain one or more i-handles of groups to which the user belongs. Finally, the version structure contains a version vector consisting of a version number for every user and group in the system.

When user u performs a file system operation, u 's client acquires the global lock and downloads the latest version structure for each user and group. We call this set of version structures the *version structure list*, or VSL. (Much of the VSL's transfer can be elided if only a few users and groups have changed version structures since the user's last operation.) The client then computes a new version structure z by potentially updating i-handles and by setting the version numbers in z to reflect the current state of the file system.

More specifically, to set the i-handles in z , on a fetch,

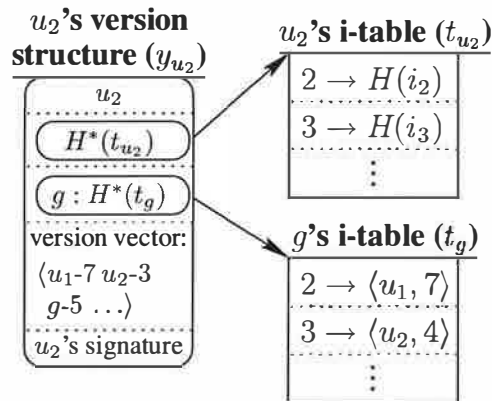


Figure 3: A version structure containing a group i-handle.

the client simply copies u 's previous i-handle into z , as nothing has changed. For a modify, the client computes and includes new i-handles for u and for any groups whose i-tables it is modifying.

The client then sets z 's version vector to reflect the version number of each VSL entry. For any version structure like z , and any principal (user or group) p , let $z[p]$ denote p 's version number in z 's version vector (or 0 if z contains no entry for p). For each principal p , if y_p is p 's entry in the VSL (i.e., the version structure containing p 's latest i-handle), set $z[p] \leftarrow y_p[p]$.

Finally, the client bumps version numbers to reflect the i-handles in z . It sets $z[u] \leftarrow z[u] + 1$, since z always

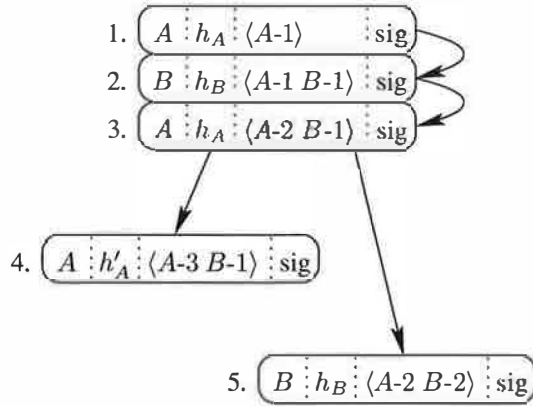


Figure 4: Signed version structures with a forking attack.

contains u 's i-handle, and for any group g whose i-handle z contains, sets $z[g] \leftarrow z[g] + 1$.

The client then checks the VSL for consistency. Given two version structures x and y , we define $x \leq y$ iff $\forall p x[p] \leq y[p]$. To check consistency, the client verifies that the VSL contains u 's previous version structure, and that the set of all VSL entries combined with z is totally ordered by \leq . If it is, the user signs the new version structure and sends it to the server with a COMMIT RPC. The server adds the new structure to the VSL and retires the old entries for updated i-handles, at which point the client releases the file system lock.

Figure 4 revisits the forking attack from the end of Section 3.1, showing how version vectors evolve in SUNDR. With each version structure signed, a user reflects the highest version number seen from every other user, and also increments his own version number to reflect the most recent i-handle. A violation of consistency causes users to sign *incompatible* version structures—i.e., two structures x and y such that $x \not\leq y$ and $y \not\leq x$. In this example, the server performs a forking attack after step 3. User A updates his i-handle from h_A to h'_A in 4, but in 5, B is not aware of the change. The result is that the two version structures signed in 4 and 5 are incompatible.

Just as in the straw-man file system, once two users have signed incompatible version structures, they will never again sign compatible ones, and thus cannot ever see each other's operations without detecting the attack (as proven in earlier work [16]).

One optimization worth mentioning is that SUNDR amortizes the cost of recomputing hash trees over several operations. As shown in Figure 5, an i-handle contains not just a hash tree root, but also a small log of changes that have been made to the i-table. The change log furthermore avoids the need for other users to fetch i-table blocks

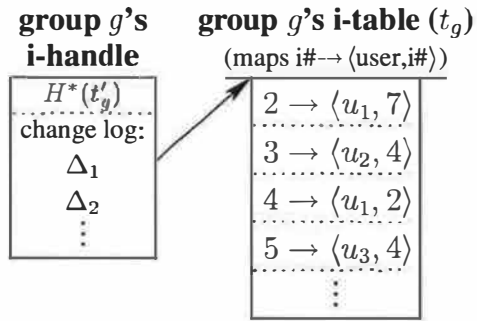


Figure 5: i-table for group g , showing the change log. t'_g is a recent i-table; applying the log to t'_g yields t_g .

when re-validating a cached file that has not changed since the hash tree root was last computed.

3.4 Concurrent SUNDR

While the version structures in SUNDR detect inconsistency, serialized SUNDR is too conservative in what it prohibits. Each client must wait for the previous client's version vector before computing and signing its own, so as to reflect the appropriate version numbers. Instead, we would like most operations to proceed concurrently. The only time one client should have to wait for another is when it reads a file the other is in the process of writing.³

3.4.1 Update certificates

SUNDR's solution to concurrent updates is for users to pre-declare a fetch or modify operation before receiving the VSL from the server. They do so with signed messages called *update certificates*. If y_u is u 's current VSL entry, an update certificate for u 's next operation contains:

- u 's next version number ($y_u[u] + 1$, unless u is pipelining multiple updates),
- a hash of u 's VSL entry ($H(y_u)$), and
- a (possibly empty) list of modifications to perform.

Each modification (or *delta*) can be one of four types:

- Set file $\langle \text{user}, i\# \rangle$ to i-hash h .
- Set group file $\langle \text{group}, i\# \rangle$ to $\langle \text{user}, i\# \rangle$.
- Set/delete entry *name* in directory $\langle \text{user}/\text{group}, i\# \rangle$.

³One might wish to avoid waiting for other clients even in the event of such a read-after-write conflict. However, this turns out to be impossible with untrusted servers. If a single signed message could atomically switch between two file states, the server could conceal the change initially, then apply it long after forking the file system, when users should no longer see each others' updates.

- Pre-allocate a range of group i-numbers (pointing them to unallocated user i-numbers).

The client sends the update certificate to the server in an UPDATE RPC. The server replies with both the VSL and a list of all pending operations not yet reflected in the VSL, which we call the *pending version list* or PVL.

Note that both fetch and modify operations require UPDATE RPCs, though fetches contain no deltas. (The RPC name refers to updating the VSL, not file contents.) Moreover, when executing complex system calls such as *rename*, a single UPDATE RPC may contain deltas affecting multiple files and directories, possibly in different i-tables.

An honest server totally orders operations according to the arrival order of UPDATE RPCs. If operation \mathcal{O}_1 is reflected in the VSL or PVL returned for \mathcal{O}_2 's UPDATE RPC, then we say \mathcal{O}_1 *happened before* \mathcal{O}_2 . Conversely, if \mathcal{O}_2 is reflected in \mathcal{O}_1 's VSL or PVL, then \mathcal{O}_2 happened before \mathcal{O}_1 . If neither happened before the other, then the server has mounted a forking attack.

When signing an update certificate, a client cannot predict the version vector of its next version structure, as the vector may depend on concurrent operations by other clients. The server, however, knows *precisely* what operations the forthcoming version structure must reflect. For each update certificate, the server therefore calculates the forthcoming version structure, except for the i-handle. This unsigned version structure is paired with its update certificate in the PVL, so that the PVL is actually a list of (update certificate, unsigned version structure) pairs.

The algorithm for computing a new version structure, z , begins as in serialized SUNDR: for each principal p , set $z[p] \leftarrow y_p[p]$, where y_p is p 's entry in the VSL. Then, z 's version vector must be incremented to reflect pending updates in the PVL, including u 's own. For user version numbers, this is simple; for each update certificate signed by user u , set $z[u] \leftarrow z[u] + 1$. For groups, the situation is complicated by the fact that operations may commit out of order when slow and fast clients update the same i-table. For any PVL entry updating group g 's i-table, we wish to increment $z[g]$ if and only if the PVL entry happened after y_g (since we already initialized $z[g]$ with $y_g[g]$). We determine whether or not to increment the version number by comparing y_g to the PVL entry's unsigned version vector, call it ℓ . If $\ell \not\leq y_g$, set $z[g] \leftarrow z[g] + 1$. The result is the same version vector one would obtain in serialized SUNDR by waiting for all previous version structures.

Upon receiving the VSL and PVL, a client ensures that the VSL, the unsigned version structures in the PVL, and its new version structure are totally ordered. It also checks for conflicts. If none of the operations in the PVL change files the client is currently fetching or group i-tables it is

modifying, the client simply signs a new version structure and sends it to the server for inclusion in the VSL.

3.4.2 Update conflicts

If a client is fetching a file and the PVL contains a modification to that file, this signifies a read-after-write conflict. In this case, the client still commits its version structure as before but then waits for fetched files to be committed to the VSL before returning to the application. (A FETCHPENDING RPC lets clients request a particular version structure from the server as soon as it arrives.)

A trickier situation occurs when the PVL contains a modification to a group i-handle that the client also wishes to modify, signifying a write-after-write conflict. How should a client, u , modifying a group g 's i-table, t_g , recompute g 's i-handle, h_g , when other operations in the PVL also affect t_g ? Since any operation in the PVL happened before u 's new version structure, call it z , the handle h_g in z must reflect all operations on t_g in the PVL. On the other hand, if the server has behaved incorrectly, one or more of the forthcoming version structures corresponding to these PVL entries may be incompatible with z . In this case, it is critical that z not somehow "launder" operations that should have alerted people to the server's misbehavior.

Recall that clients already check the PVL for read-after-write conflicts. When a client sees a conflicting modification in the PVL, it will wait for the corresponding VSL entry even if u has already incorporated the change in h_g . However, the problem remains that a malicious server might prematurely drop entries from the PVL, in which case a client could incorrectly fetch modifications reflected by t_g but never properly committed.

The solution is for u to incorporate any modifications of t_g in the PVL not yet reflected in y_g , and also to record the current contents of the PVL in a new field of the version structure. In this way, other clients can detect missing PVL entries when they notice those entries referenced in u 's version structure. Rather than include the full PVL, which might be large, u simply records, for each PVL entry, the user performing the operation, that user's version number for the operation, and a hash of the expected version structure with i-handles omitted.

When u applies changes from the PVL, it can often do so by simply appending the changes to the change log of g 's i-handle, which is far more efficient than rehashing the i-table and often saves u from fetching uncached portions of the i-table.

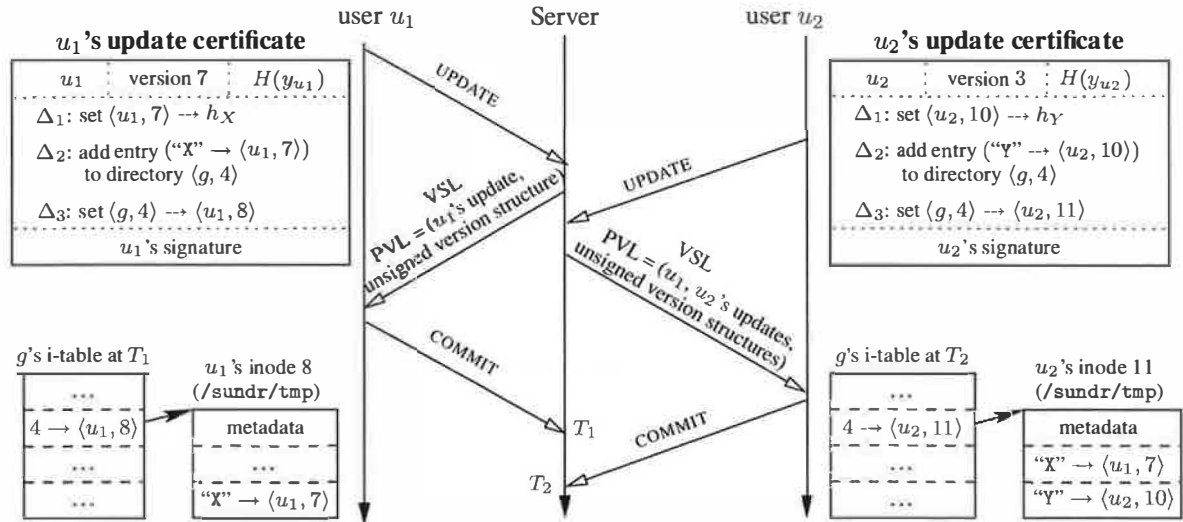


Figure 6: Concurrent updates to /sundr/tmp/ by different users.

3.4.3 Example

Figure 6 shows an example of two users u_1 and u_2 in the group g modifying the same directory. u_1 creates file X while u_2 creates Y , both in /sundr/tmp/. The directory is group-writable, while the files are not. (For the example, we assume no other pending updates.)

Assume /sundr/tmp/ is mapped to group g 's i-number 4. User u_1 first calculates the i-hash of file X , call it h_X , then allocates his own i-number for X , call it 7. u_1 then allocates another i-number, 8, to hold the contents of the modified directory. Finally, u_1 sends the server an update certificate declaring three deltas, namely the mapping of file ⟨ u_1 , 7⟩ to i-hash h_X , the addition of entry ("X" → ⟨ u_1 , 7⟩) to the directory, and the re-mapping of g 's i-number 4 to ⟨ u_1 , 8⟩.

u_2 similarly sends the server an update certificate for the creation of file Y in /sundr/tmp/. If the server orders u_1 's update before u_2 's, it will respond to u_1 with the VSL and a PVL containing only u_1 's update, while it will send u_2 a PVL reflecting both updates. u_2 will therefore apply u_1 's modification to the directory before computing the i-handle for g , incorporating u_1 's directory entry for X . u_2 would also ordinarily incorporate u_1 's re-mapping of the directory ⟨ g , 4⟩ → ⟨ u_1 , 7⟩, except that u_2 's own re-mapping of the same directory supersedes u_1 's.

An important subtlety of the protocol, shown in Figure 7, is that u_2 's version structure contains a hash of u_1 's forthcoming version structure (without i-handles). This ensures that if the server surreptitiously drops u_1 's update certificate from the PVL before u_1 commits, whoever sees the incorrect PVL must be forked from both u_1 and u_2 .

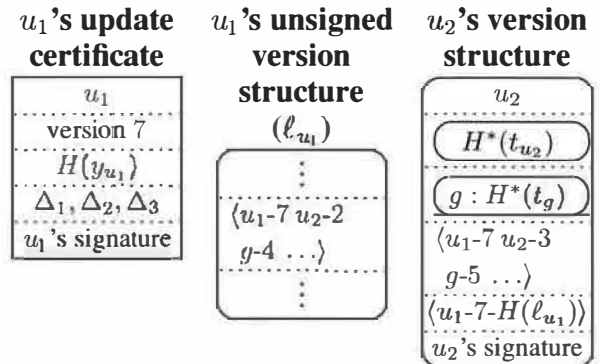


Figure 7: A pending update by user u_1 , reflected in user u_2 's version structure.

4 Discussion

SUNDR only detects attacks; it does not resolve them. Following a server compromise, two users might find themselves caching divergent copies of the same directory tree. Resolving such differences has been studied in the context of optimistic file system replication [13, 22], though invariably some conflicts require application-specific reconciliation. With CVS, users might employ CVS's own merging facilities to resolve forks.

SUNDR's protocol leaves considerable opportunities for compression and optimization. In particular, though version structure signatures must cover a version vector with all users and groups, there is no need to transmit entire vectors in RPCs. By ordering entries from most- to

least-recently updated, the tail containing idle principals can be omitted on all but a client's first UPDATE RPC. Moreover, by signing a hash of the version vector and hashing from oldest to newest, clients could also pre-hash idle principals' version numbers to speed version vector signatures. Finally, the contents of most unsigned version structures in the PVL is implicit based on the order of the PVL and could be omitted (since the server computes unsigned version structures deterministically based on the order in which it receives UPDATE RPCs). None of these optimizations is currently implemented.

SUNDR's semantics differ from those of traditional Unix. Clients supply file modification and inode change times when modifying files, allowing values that might be prohibited in Unix. There is no time of last access. Directories have no "sticky bit." A group-writable file in SUNDR is not owned by a user (as in Unix) but rather is owned by the group; such a file's "owner" field indicates the last user who wrote to it. In contrast to Unix disk quotas, which charge the owner of a group-writable file for writes by other users, if SUNDR's block store enforced quotas, they would charge each user for precisely the blocks written by that user.

One cannot change the owner of a file in SUNDR. However, SUNDR can copy arbitrarily large files at the cost of a few pointer manipulations, due to its hash-based storage mechanism. Thus, SUNDR implements *chown* by creating a copy of the file owned by the new user or group and updating the directory entry to point to the new copy. Doing so requires write permission on the directory and changes the semantics of hard links (since *chown* only affects a single link).

Yet another difference from Unix is that the owner of a directory can delete any entries in the directory, including non-empty subdirectories to which he or she does not have write permission. Since Unix already allows users to rename such directories away, additionally allowing delete permission does not appreciably affect security. In a similar vein, users can create multiple hard links to directories, which could confuse some Unix software, or could be useful in some situations. Other types of malformed directory structure are interpreted as equivalent to something legal (e.g., only the first of two duplicate directory entries counts).

SUNDR does not yet offer read protection or confidentiality. Confidentiality can be achieved through encrypted storage, a widely studied problem [5, 10, 12, 34].

In terms of network latency, SUNDR is comparable with other polling network file systems. SUNDR waits for an UPDATE RPC to complete before returning from an application file system call. If the system call caused only

modifies, or if all fetched data hit in the cache, this is the only synchronous round trip required; the COMMIT can be sent in the background (except for *fsync*). This behavior is similar to systems such as NFS3, which makes an ACCESS RPC on each open and writes data back to the server on each close. We note that callback- or lease-based file systems can actually achieve zero round trips when the server has committed to notifying clients of cache invalidations.

5 File system implementation

The SUNDR client is implemented at user level, using a modified version of the *xfs* device driver from the ARLA file system [33] on top of a slightly modified FreeBSD kernel. Server functionality is divided between two programs, a consistency server, which handles update certificates and version structures, and a block store, which actually stores data, update certificates, and version structures on disk. For experiments in this paper, the block server and consistency server ran on the same machine, communicating over Unix-domain sockets. They can also be configured to run on different machines and communicate over an authenticated TCP connection.

5.1 File system client

The *xfs* device driver used by SUNDR is designed for whole-file caching. When a file is opened, *xfs* makes an upcall to the SUNDR client asking for the file's data. The client returns the identity of a local file that has a cached copy of the data. All reads and writes are performed on the cached copy, without further involvement of SUNDR. When the file is closed (or flushed with *fsync*), if it has been modified, *xfs* makes another upcall asking the client to write the data back to the server. Several other types of upcalls allow *xfs* to look up names in directories, request file attributes, create/delete files, and change metadata.

As distributed, *xfs*'s interface posed two problems for SUNDR. First, *xfs* caches information like local file bindings to satisfy some requests without upcalls. In SUNDR, some of these requests require interaction with the consistency server for the security properties to hold. We therefore modified *xfs* to invalidate its cache tokens immediately after getting or writing back cached data, so as to ensure that the user-level client gets control whenever the protocol requires an UPDATE RPC. We similarly changed *xfs* to defeat the kernel's name cache.

Second, some system calls that should require only a single interaction with the SUNDR consistency server result in multiple kernel vnode operations and *xfs* upcalls. For example, the system call "`stat ("a/b/c", &sb)`"

results in three `xfs` `GETNODE` upcalls (for the directory lookups) and one `GETATTR`. The whole system call should require only one `UPDATE` RPC. Yet if the user-level client does not know that the four upcalls are on behalf of the same system call, it must check the freshness of its i-handles four separate times with four `UPDATE` RPCs.

To eliminate unnecessary RPCs, we modified the FreeBSD kernel to count the number of system call invocations that might require an interaction with the consistency server. We increment the counter at the start of every system call that takes a pathname as an argument (e.g., `stat`, `open`, `readlink`, `chdir`). The SUNDR client memory-maps this counter and records the last value it has seen. If `xfs` makes an upcall that does not change the state of the file system, and the counter has not changed, then the client can use its cached copies of all i-handles.

5.2 Signature optimization

The cost of digital signatures on the critical path in SUNDRA is significant. Our implementation therefore uses the ESIGN signature scheme,⁴ which is over an order of magnitude faster than more popular schemes such as RSA. All experiments reported in this paper use 2,048-bit public keys, which, with known techniques, would require a much larger work factor to break than 1,024-bit RSA.

To move verification out of the critical path, the consistency server also processes and replies to an `UPDATE` RPC before verifying the signature on its update certificate. It verifies the signature after replying, but before accepting any other RPCs from other users. If the signature fails to verify, the server removes the update certificate from the PVL and drops the TCP connection to the forging client. (Such behavior is acceptable because only a faulty client would send invalid signatures.) This optimization allows the consistency server's verification of one signature to overlap with the client's computation of the next.

Clients similarly overlap computation and network latency. Roughly half the cost of an ESIGN signature is attributable to computations that do not depend on the message contents. Thus, while waiting for the reply to an `UPDATE` RPC, the client precomputes its next signature.

5.3 Consistency server

The consistency server orders operations for SUNDRA clients and maintains the VSL and PVL as described in Section 3. In addition, it polices client operations and rejects invalid RPCs, so that a malicious user cannot cause

⁴Specifically, we use the version of ESIGN shown secure in the random oracle model by [21], with parameter $e = 8$.

an honest server to fail. For crash recovery, the consistency server must store VSL and PVL to persistent storage *before* responding to client RPCs. The current consistency server stores these to the block server. Because the VSLs and PVLs are small relative to the size of the file system, it would also be feasible to use non-volatile RAM (NVRAM).

6 Block store implementation

A block storage daemon called *bstor* handles all disk storage in SUNDRA. Clients interact directly with *bstor* to store blocks and retrieve them by SHA-1 hash value. The consistency server uses *bstor* to store signed update and version structures. Because a SUNDRA server does not have signature keys, it lacks permission to repair the file system after a crash. For this reason, *bstor* must synchronously store all data to disk before returning to clients, posing a performance challenge. *bstor* therefore heavily optimizes synchronous write performance.

bstor's basic idea is to write incoming data blocks to a temporary log, then to move these blocks to Venti-like storage in batches. Venti [24] is an archival block store that appends variable-sized blocks to a large, append-only IDE log disk while indexing the blocks by SHA-1 hash on one or more fast SCSI disks. *bstor*'s temporary log relaxes the archival semantics of Venti, allowing short-lived blocks to be deleted within a small window of their creation. *bstor* maintains an archival flavor, though, by supporting periodic file system snapshots.

The temporary log allows *bstor* to achieve low latency on synchronous writes, which under Venti require an index lookup to ensure the block is not a duplicate. Moreover, *bstor* sector-aligns all blocks in the temporary log, temporarily wasting an average of half a sector per block so as to avoid multiple writes to the same sector, which would each cost at least one disk rotation. The temporary log improves write throughput even under sustained load, because transferring blocks to the permanent log in large batches allows *bstor* to order index disk accesses.

bstor keeps a large in-memory cache of recently used blocks. In particular, it caches all blocks in the temporary log so as to avoid reading from the temporary log disk. Though *bstor* does not currently use special hardware, in Section 7 we describe how SUNDRA's performance would improve if *bstor* had a small amount of NVRAM to store update certificates.

6.1 Interface

bstor exposes the following RPCs to SUNDRA clients:

```

STORE (header, block)
RETRIEVE (hash)
VSTORE (header, pubkey, n, block)
VRETRIEVE (pubkey, n, [time])
DECREF (hash)
SNAPSHOT ()

```

The STORE RPC writes a block and its header to stable storage if *bstor* does not already have a copy of the block. The header has information encapsulating the block's owner and creation time, as well as fields useful in concert with encoding or compression. The RETRIEVE RPC retrieves a block from the store given its SHA-1 hash. It also returns the first header STORED with the particular block.

The VSTORE and VRETRIEVE RPCs are like STORE and RETRIEVE, but for signed blocks. Signed blocks are indexed by the public key and a small index number, *n*. VRETRIEVE, by default, fetches the most recent version of a signed block. When supplied with a timestamp as an optional third argument, VRETRIEVE returns the newest block written before the given time.

DECREF (short for “decrement reference count”) informs the store that a block with a particular SHA-1 hash might be discarded. SUNDR clients use DECREF to discard temporary files and short-lived metadata. *bstor*'s deletion semantics are conservative. When a block is first stored, *bstor* establishes a short window (one minute by default) during which it can be deleted. If a client STORES then DECREFS a block within this window, *bstor* marks the block as garbage and does not permanently store it. If two clients store the same block during the dereference window, the block is marked as permanent.

An administrator should issue a SNAPSHOT RPC periodically to create a coherent file system image that clients can later revert to in the case of accidental data disruption. Upon receiving this RPC, *bstor* simply immunizes all newly-stored blocks from future DECREF's and flags them to be stored in the permanent log. SNAPSHOT and VRETRIEVE's *time* argument are designed to allow browsing of previous file system state, though this functionality is not yet implemented in the client.

6.2 Index

bstor's index system locates blocks on the permanent log, keyed by their SHA-1 hashes. An ideal index is a simple in-memory hash table mapping 20-byte SHA-1 block hashes to 8-byte log disk offsets. If we assume that the average block stored on the system is 8 KB, then the index must have roughly 1/128 the capacity of the log disk. Although at present such a ratio of disk to memory is pos-

sible with commodity components, we are not convinced that memory will keep up with hard disks in the future.

We instead use Venti's strategy of striping a disk-resident hash table over multiple high-speed SCSI disks. *bstor* hashes 20-byte SHA-1 hashes down to $\langle \text{index-disk-id}, \text{index-disk-offset} \rangle$ pairs. The disk offsets point to sector-sized on-disk data structures called *buckets*, which contain 15 *index-entries*, sorted by SHA-1 hash. *index-entries* in turn map SHA-1 hashes to offsets on the permanent data log. Whenever an index-entry is written to or read from disk, *bstor* also stores it in an in-memory LRU cache.

bstor accesses the index system as Venti does when answering RETRIEVE RPCs that miss the block cache. When *bstor* moves data from the temporary to the permanent log, it must access the index system sometimes twice per block (once to check a block is not a duplicate, and once to write a new index entry after the block is committed to the permanent log). In both cases, *bstor* sorts these disk accesses so that the index disks service a batch of requests with one disk arm sweep. Despite these optimizations, *bstor* writes blocks to the permanent log in the order they arrived; randomly reordering blocks would hinder sequential read performance over large files.

6.3 Data management

To recover from a crash or an unclean shutdown, the system first recreates an index consistent with the permanent log, starting from its last known checkpoint. Index recovery is necessary because the server updates the index lazily after storing blocks to the permanent log. *bstor* then processes the temporary log, storing all fresh blocks to the permanent log, updating the index appropriately.

Venti's authors argue that archival storage is practical because IDE disk capacity is growing faster than users generate data. For users who do not fit this paradigm, however, *bstor* could alternatively be modified to support mark-and-sweep garbage collection. The general idea is to copy all reachable blocks to a new log disk, then recycle the old disk. With two disks, *bstor* could still respond to RPCs during garbage collection.

7 Performance

The primary goal in testing SUNDR was to ensure that its security benefits do not come at too high a price relative to existing file systems. In this section, we compare SUNDR's overall performance to NFS. We also perform microbenchmarks to help explain our application-level re-

sults, and to support our claims that our block server outperforms a Venti-like architecture in our setting.

7.1 Experimental setup

We carried out our experiments on a cluster of 3 GHz Pentium IV machines running FreeBSD 4.9. All machines were connected with fast Ethernet with ping times of 110 μ s. For block server microbenchmarks, we additionally connected the block server and client with gigabit Ethernet. The machine running *bstor* has 3 GB of RAM and an array of disks: four Seagate Cheetah 18 GB SCSI drives that spin at 15,000 RPM were used for the index; two Western Digital Caviar 180 GB 7200 RPM EIDE drives were used for the permanent and temporary logs.

7.2 Microbenchmarks

7.2.1 *bstor*

Our goals in evaluating *bstor* are to quantify its raw performance and justify our design improvements relative to Venti. In our experiments, we configured *bstor*'s four SCSI disks each to use 4 GB of space for indexing. If one hopes to maintain good index performance (and not overflow buckets), then the index should remain less than half full. With our configuration (8 GB of usable index and 32-byte index entries), *bstor* can accommodate up to 2 TB of permanent data. For flow control and fairness, *bstor* allowed clients to make up to 40 outstanding RPCs. For the purposes of the microbenchmarks, we disabled *bstor*'s block cache but enabled an index cache of up to 100,000 entries. The circular temporary log was 720 MB and never filled up during our experiments.

We measured *bstor*'s performance while storing and fetching a batch of 20,000 unique 8 KB blocks. Figure 8 shows the averaged results from 20 runs of a 20,000 block experiment. In all cases, standard deviations were less than 5% of the average results. The first two results show that *bstor* can absorb bursts of 8 KB blocks at almost twice fast Ethernet rates, but that sustained throughput is limited by *bstor*'s ability to shuffle blocks from the temporary to the permanent logs, which it can do at 11.9 MB/s. The bottleneck in STOREing blocks to the temporary log is currently CPU, and future versions of *bstor* might eliminate some unnecessary *memcpy*s to achieve better throughput. On the other hand, *bstor* can process the temporary log only as fast as it can read from its index disks, and there is little room for improvement here unless disks become faster or more index disks are used.

To compare with a Venti-like system, we implemented a Venti-like store mechanism. In VENTL_STORE, *bstor*

Operation	MB/s
STORE (burst)	18.4
STORE (sustained)	11.9
VENTL_STORE	5.1
RETRIEVE (random + cold index cache)	1.2
RETRIEVE (sequential + cold index cache)	9.1
RETRIEVE (sequential + warm index cache)	25.5

Figure 8: *bstor* throughput measurements with the block cache disabled.

first checks for a block's existence in the index and stores the block to the permanent log only if it is not found. That is, each VENTL_STORE entails an access to the index disks. Our results show that VENTL_STORE can achieve only 27% of STORE's burst throughput, and 43% of its sustained throughput.

Figure 8 also presents read measurements for *bstor*. If a client reads blocks in the same order they are written (i.e., "sequential" reads), then *bstor* need not seek across the permanent log disk. Throughput in this case is limited by the per-block cost of locating hashes on the index disks and therefore increases to 25.5 MB/s with a warm index cache. Randomly-issued reads fare poorly, even with a warm index cache, because *bstor* must seek across the permanent log. In the context of SUNDR, slow random RETRIEVES should not affect overall system performance if the client aggressively caches blocks and reads large files sequentially.

Finally, the latency of *bstor* RPCs is largely a function of seek times. STORE RPCs do not require seeks and therefore return in 1.6 ms. VENTL_STORE returns in 6.7 ms (after one seek across the index disk at a cost of about 4.4 ms). Sequential RETRIEVES that hit and miss the index cache return in 1.9 and 6.3 ms, respectively. A seek across the log disk takes about 6.1 ms; therefore random RETRIEVES that hit and miss the index cache return in 8.0 and 12.4 ms respectively.

7.2.2 Cryptographic overhead

SUNDR clients sign and verify version structures and update certificates using 2,048-bit ESIGN keys. Our implementation (based on the GNU Multiprecision library version 4.1.4) can complete signatures in approximately 150 μ s and can verify them 100 μ s. Precomputing a signature requires roughly 80 μ s, while finalizing a precomputed signature is around 75 μ s. We observed that these measurements can vary on the Pentium IV by as much as a factor of two, even in well-controlled micro-benchmarks. By comparison, an optimized version of the Rabin signature scheme with 1,280-bit keys, running on the same

hardware, can compute signatures in 3.1 ms and can verify them in 27 μ s.

7.3 End-to-end evaluation

In end-to-end experiments, we compare SUNDR to both NFS2 and NFS3 servers running on the same hardware. To show NFS in the best possible light, the NFS experiments run on the fast SCSI disks SUNDR uses for indexes, not the slower, larger EIDE log disks. We include NFS2 results because NFS2's write-through semantics are more like SUNDR's. Both NFS2 and SUNDR write all modified file data to disk before returning from a *close* system call, while NFS3 does not offer this guarantee.

Finally, we described in Section 5.3 that SUNDR clients must wait for the consistency server to write small pieces of data (VSLs and PVLs) to stable storage. The consistency server's storing of PVLs in particular is on the client's critical path. We present result sets for consistency servers running with and without flushes to secondary storage. We intend the mode with flushes disabled to simulate a consistency server with NVRAM.

All application results shown are the average of three runs. Relative standard deviations are less than 8% unless otherwise noted.

7.3.1 LFS small file benchmark

The LFS small file benchmark [28] tests SUNDR's performance on simple file system operations. This benchmark creates 1,000 1 KB files, reads them back, then deletes them. We have modified the benchmark slightly to write random data to the 1 KB files; writing the same file 1,000 times would give SUNDR's hash-based block store an unfair advantage.

Figure 9 details our results when only one client is accessing the file system. In the **create** phase of the benchmark, a single file creation entails system calls to *open*, *read* and *close*. On SUNDR/NVRAM, the *open* call involves two serialized rounds of the consistency protocol, each of which costs about 2 ms; the *write* call is a no-op, since file changes are buffered until *close*; and the *close* call involves one round of the protocol and one synchronous write of file data to the block server, which the client can overlap. Thus, the entire sequence takes about 6 ms. Without NVRAM, each round of the protocol takes approximately 1-2 ms longer, because the consistency server must wait for *bstor* to flush.

Unlike SUNDR, an NFS server must wait for at least one disk seek when creating a new file because it synchronously writes metadata. A seek costs at least 4 ms on our fast SCSI drives, and thus NFS can do no better than

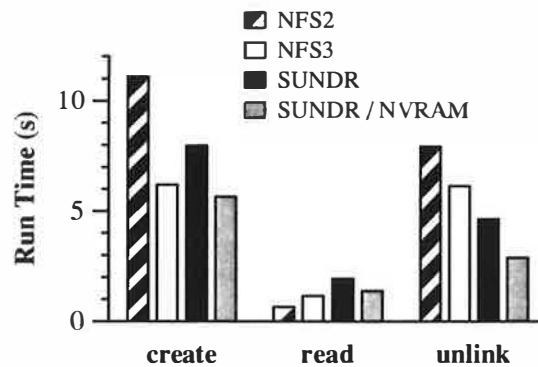


Figure 9: Single client LFS Small File Benchmark. 1000 operations on files with 1 KB of random content.

4 ms per file creation. In practice, NFS requires about 6 ms to service the three system calls in the **create** stage.

In the **read** phase of the benchmark, SUNDR performs one round of the consistency protocol in the *open* system call. The NFS3 client still accesses the server with an ACCESS RPC, but the server is unlikely to need any data not in its buffer cache at this point, and hence no seeking is required. NFS2 does not contact the server in this phase.

In the **unlink** stage of the benchmark, clients issue a single *unlink* system call per file. An *unlink* for SUNDR triggers one round of the consistency protocol and an asynchronous write to the block server to store updated i-table and directory blocks. SUNDR and SUNDR/NVRAM in particular can outperform NFS in this stage of the experiment because NFS servers again require at least one synchronous disk seek per file *unlinked*.

We also performed experiments with multiple clients performing the LFS small file benchmark concurrently in different directories. Results for the **create** phase are reported in Figure 10 and the other phases of the benchmark show similar trends. A somewhat surprising result is that SUNDR actually scales better than NFS as client concurrency increases in our limited tests. NFS is seek-bound even in the single client case, and the number of seeks the NFS servers require scale linearly with the number of concurrent clients. For SUNDR, latencies induced by the consistency protocol limit individual client performance, but these latencies overlap when clients act concurrently. SUNDR's disk accesses are also scalable because they are sequential, sector-aligned writes to *bstor*'s temporary log.

7.3.2 Group contention

The group protocol incurs additional overhead when folding other users' changes into a group i-table or directory. We characterized the cost of this mechanism by measur-

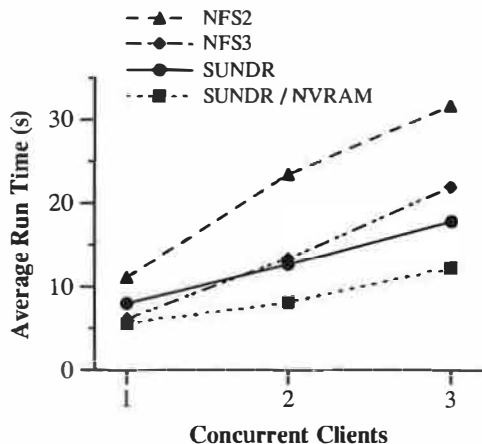


Figure 10: Concurrent LFS Small File Benchmark, **create** phase. 1000 creations of 1 KB files. (Relative standard deviation for SUNDR in 3 concurrent clients case is 13.7%)

ing a workload with a high degree of contention for a group-owned directory. We ran a micro-benchmark that simultaneously created 300 new files in the same, group-writable directory on two clients. Each concurrent create required the client to re-map the group i-number in the group i-table and apply changes to the user's copy of the directory.

The clients took an average of 4.60 s and 4.26 s on SUNDR/NVRAM and NFS3 respectively. For comparison, we also ran the benchmark concurrently in two separate directories, which required an average of 2.94 s for SUNDR/NVRAM and 4.05 s for NFS3. The results suggests that while contention incurs a noticeable cost, SUNDR's performance even in this case is not too far out of line with NFS3.

7.3.3 Real workloads

Figure 11 shows SUNDR's performance in untaring, configuring, compiling, installing and cleaning an emacs 20.7 distribution. During the experiment, the SUNDR client sent a total of 42,550 blocks to the block server, which totaled 139.24 MB in size. Duplicate blocks, which *bstor* discards, account for 29.5% of all data sent. The client successfully DECFEd 10,747 blocks, for a total space savings of 11.3%. In the end, 25,740 blocks which totaled 82.21 MB went out to permanent storage.

SUNDR is faster than NFS2 and competitive with NFS3 in most stages of the Emacs build process. We believe that SUNDR's sluggish performance in the *install* phase is an artifact of our implementation, which serializes concurrent *xfs* upcalls for simplicity (and not correct-

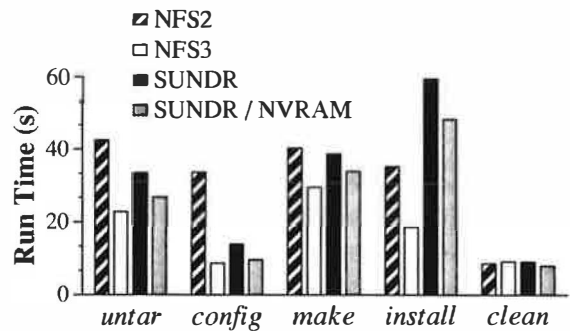


Figure 11: Installation procedure for emacs_20.7

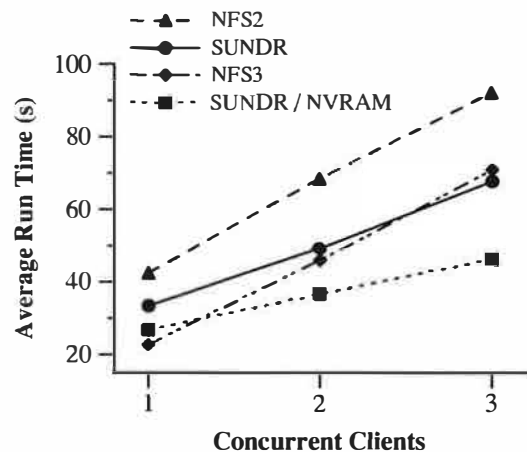


Figure 12: Concurrent *untar* of emacs_20.7.tar

ness). Concurrent *xfs* upcalls are prevalent in this phase of the experiment due to the *install* command's manipulation of file attributes.

Figure 12 details the performance of the *untar* phase of the Emacs build as client concurrency increases. We noted similar trends for the other phases of the build process. These experiments suggest that the scalability SUNDR exhibited in the LFS small file benchmarks extends to real file system workloads.

7.3.4 CVS on SUNDR

We tested CVS over SUNDR to evaluate SUNDR's performance as a source code repository. Our experiment follows a typical progression. First, client *A* imports an arbitrary source tree—in this test *groff-1.17.2*, which has 717 files totaling 6.79 MB. Second, clients *A* and *B* check out a copy to their local disks. Third, *A* commits *groff-1.18*, which affects 549 files (6.06 MB). Lastly, *B* updates its local copy. Figure 13 shows the results.

SUNDR fares badly on the commit phase because CVS repeatedly opens, memory maps, unmaps, and closes each

Phase	SUNDR	SUNDR NVRAM	NFS3	SSH
Import	13.0	10.0	4.9	7.0
Checkout	13.5	11.5	11.6	18.2
Commit	38.9	32.8	15.7	11.5
Update	19.1	15.9	13.3	11.5

Figure 13: Run times for CVS experiments (in seconds).

repository file several times in rapid succession. Every open requires an iteration of the consistency protocol in SUNDR, while FreeBSD's NFS3 apparently elides or asynchronously performs ACCESS RPCs after the first of several closely-spaced *open* calls. CVS could feasibly cache memory-mapped files at this point in the experiment, since a single CVS client holds a lock on the directory. This small change would significantly improve SUNDR's performance in the benchmark.

8 Related work

A number of non-networked file systems have used cryptographic storage to keep data secret [5, 34] and check integrity [31]. Several network file systems provide varying degrees integrity checks but reduce integrity on read sharing [25] or are vulnerable to consistency attacks [10, 12, 19]. SUNDR is the first system to provide well-defined consistency semantics for an untrusted server. An unimplemented but previously published version of the SUNDR protocol [16] had no groups and thus did not address write-after-write conflicts.

The Byzantine fault-tolerant file system, BFS [6], uses replication to ensure the integrity of a network file system. As long as more than 2/3 of a server's replicas are uncompromised, any data read from the file system will have been written by a legitimate user. SUNDR, in contrast, does not require any replication or place any trust in machines other than a user's client. However, SUNDR provides weaker freshness guarantees than BFS, because of the possibility that a malicious SUNDR server can fork the file system state if users have no other evidence of each other's on-line activity.

Several projects have investigated storing file systems on peer-to-peer storage systems comprised of potentially untrusted nodes. Farsite [3] spreads such a file system across people's unreliable desktop machines. CFS [7] is a secure read-only file P2P system. Ivy [20], a read-write version of CFS, can be convinced to re-order operations clients have already seen. Pond [27] relies on a trusted "inner core" of machines for security, distributing trust in a BFS-like way.

SUNDR uses hash trees, introduced in [18], to verify a

file block's integrity without touching the entire file system. Duchamp [8], BFS [6], SFSRO [9] and TDB [14] have all made use of hash trees for comparing data or checking the integrity of part of a larger collection of data.

SUNDR uses version vectors to detect consistency violations. Version vectors were used by Ficus [22] to detect update conflicts between file system replicas, and have also been used to secure partial orderings [26, 30]. Our straw-man file system somewhat resembles timeline entanglement [15], which reasons about the temporal ordering of system states using hash chains.

9 Conclusions

SUNDR is a general-purpose, multi-user network file system that never presents applications with incorrect file system state, even when the server has been compromised. SUNDR's protocol provably guarantees fork consistency, which essentially ensures that the server either behaves correctly or that its failure will be detected after communication among users. In any event, the consequences of an undetected server compromise are limited to concealing users' operations from each other after some forking point; the server cannot tamper with, inject, re-order, or suppress file writes in any other way.

Measurements of our implementation show performance that is usually close to and sometimes better than the popular NFS file system. Yet by reducing the amount of trust placed in the server, SUNDR both increases people's options for managing data and significantly improves the security of their files.

Acknowledgments

Thanks to Michael Freedman, Kevin Fu, Daniel Giffin, Frans Kaashoek, Jinyang Li, Robert Morris, the anonymous reviewers, and our shepherd Jason Flinn.

This material is based upon work supported by the National Science Foundation (NSF) under grant CCR-0093361. Maxwell Krohn is partially supported by an NSF Graduate Fellowship, David Mazières by an Alfred P. Sloan research fellowship, and Dennis Shasha by NSF grants IIS-9988636, MCB-0209754, and MCB-0115586.

References

- [1] Apache.org compromise report. <http://www.apache.org/info/20010519-hack.html>, May 2001.
- [2] Debian investigation report after server compromises. <http://www.debian.org/News/2003/20031202>, December 2003.

- [3] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 1–14, December 2002.
- [4] Brian Berliner. CVS II: Parelizing software development. In *Proceedings of the Winter 1990 USENIX*, Colorado Springs, CO, 1990. USENIX.
- [5] Matt Blaze. A cryptographic file system for unix. In *1st ACM Conference on Communications and Computing Security*, pages 9–16, November 1993.
- [6] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 173–186, New Orleans, LA, February 1999.
- [7] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 202–215, Chateau Lake Louise, Banff, Canada, October 2001. ACM.
- [8] Dan Duchamp. A toolkit approach to partially disconnected operation. In *Proceedings of the 1997 USENIX*, pages 305–318. USENIX, January 1997.
- [9] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. *ACM Transactions on Computer Systems*, 20(1):1–24, February 2002.
- [10] Eu-Jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. SiRiUS: Securing Remote Untrusted Storage. In *Proceedings of the Tenth Network and Distributed System Security (NDSS) Symposium*, pages 131–145. Internet Society (ISOC), February 2003.
- [11] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages Systems*, 12(3):463–492, 1990.
- [12] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *2nd USENIX conference on File and Storage Technologies (FAST '03)*, San Francisco, CA, April 2003.
- [13] James J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, 1992.
- [14] Umesh Maheshwari and Radek Vingralek. How to build a trusted database system on untrusted storage. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, San Diego, October 2000.
- [15] Petros Maniatis and Mary Baker. Secure history preservation through timeline entanglement. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, August 2002.
- [16] David Mazières and Dennis Shasha. Building secure file systems out of Byzantine storage. In *Proceedings of the 21st Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 108–117, July 2002.
- [17] David Mazières and Dennis Shasha. Building secure file systems out of Byzantine storage. Technical Report TR2002–826, NYU Department of Computer Science, May 2002.
- [18] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology—CRYPTO '87*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378, Berlin, 1987. Springer-Verlag.
- [19] Ethan Miller, Darrell Long, William Freeman, and Benjamin Reed. Strong security for distributed file systems. In *Proceedings of the 20th IEEE International Performance, Computing, and Communications Conference*, pages 34–40, Phoenix, AZ, April 2001.
- [20] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 31–44, December 2002.
- [21] Tatsuki Okamoto and Jacques Stern. Almost uniform density of power residues and the provable security of ESIGN. In *Advances in Cryptology—ASIACRYPT*, pages 287–301, 2003.
- [22] T. W. Page, Jr., R. G. Guy, J. S. Heidemann, D. H. Ratner, P. L. Reiher, A. Goel, G. H. Kuenning, and G. J. Popek. Perspectives on optimistically replicated peer-to-peer filing. *Software Practice and Experience*, 28(2):155–180, February 1998.
- [23] D. Stott Parker, Jr., Gerald J. Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, SE-9(3):240–247, May 1983.
- [24] Sean Quinlan and Sean Dorward. Venti: a new approach to archival storage. In *First USENIX conference on File and Storage Technologies (FAST '02)*, Monterey, CA, January 2002.
- [25] David Reed and Liba Svobodova. Swallow: A distributed data storage system for a local network. In A. West and P. Janson, editors, *Local Networks for Computer Communications*, pages 355–373. North-Holland Publ., Amsterdam, 1981.
- [26] Michael Reiter and Li Gong. Securing causal relationships in distributed systems. *The Computer Journal*, 38(8):633–642, 1995.
- [27] Sean Rhea, Patrick Eaton, and Dennis Geels. Pond: The OceanStore prototype. In *2nd USENIX conference on File and Storage Technologies (FAST '03)*, San Francisco, CA, April 2003.
- [28] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 1–15, Pacific Grove, CA, October 1991. ACM.
- [29] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer 1985 USENIX*, pages 119–130, Portland, OR, 1985. USENIX.
- [30] Sean W. Smith and J. D. Tygar. Security and privacy for partial order time. In *Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems*, pages 70–79, Las Vegas, NV, October 1994.
- [31] Christopher A. Stein, John H. Howard, and Margo I. Seltzer. Unifying file system protection. In *Proceedings of the 2001 USENIX*. USENIX, June 2001.
- [32] Owen Taylor. Intrusion on www.gnome.org. <http://mail.gnome.org/archives/gnome-announce-list/2004-March/msg00114.html>, March 2004.
- [33] Assar Westerlund and Johan Danielsson. Arla—a free AFS client. In *Proceedings of the 1998 USENIX, Freenix track*, New Orleans, LA, June 1998. USENIX.
- [34] Charles P. Wright, Michael Martino, and Erez Zadok. NCryptfs: A secure and convenient cryptographic file system. In *Proceedings of the Annual USENIX Technical Conference*, pages 197–210, June 2003.
- [35] Tatu Ylönen. SSH – secure login connections over the Internet. In *Proceedings of the 6th USENIX Security Symposium*, pages 37–42, San Jose, CA, July 1996.

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

for a rewrite of our production indexing system. Section 7 discusses related and future work.

2 Programming Model

The computation takes a set of *input* key/value pairs, and produces a set of *output* key/value pairs. The user of the MapReduce library expresses the computation as two functions: *Map* and *Reduce*.

Map, written by the user, takes an input pair and produces a set of *intermediate* key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key *I* and passes them to the *Reduce* function.

The *Reduce* function, also written by the user, accepts an intermediate key *I* and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per *Reduce* invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

2.1 Example

Consider the problem of counting the number of occurrences of each word in a large collection of documents. The user would write code similar to the following pseudo-code:

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");

reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```

The map function emits each word plus an associated count of occurrences (just '1' in this simple example). The reduce function sums together all counts emitted for a particular word.

In addition, the user writes code to fill in a *mapreduce specification* object with the names of the input and output files, and optional tuning parameters. The user then invokes the *MapReduce* function, passing it the specification object. The user's code is linked together with the MapReduce library (implemented in C++). Appendix A contains the full program text for this example.

2.2 Types

Even though the previous pseudo-code is written in terms of string inputs and outputs, conceptually the map and reduce functions supplied by the user have associated types:

```
map      (k1, v1)          → list(k2, v2)
reduce   (k2, list(v2))    → list(v2)
```

I.e., the input keys and values are drawn from a different domain than the output keys and values. Furthermore, the intermediate keys and values are from the same domain as the output keys and values.

Our C++ implementation passes strings to and from the user-defined functions and leaves it to the user code to convert between strings and appropriate types.

2.3 More Examples

Here are a few simple examples of interesting programs that can be easily expressed as MapReduce computations.

Distributed Grep: The map function emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.

Count of URL Access Frequency: The map function processes logs of web page requests and outputs $\langle \text{URL}, 1 \rangle$. The reduce function adds together all values for the same URL and emits a $\langle \text{URL}, \text{total count} \rangle$ pair.

Reverse Web-Link Graph: The map function outputs $\langle \text{target}, \text{source} \rangle$ pairs for each link to a target URL found in a page named source. The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: $\langle \text{target}, \text{list}(\text{source}) \rangle$

Term-Vector per Host: A term vector summarizes the most important words that occur in a document or a set of documents as a list of $\langle \text{word}, \text{frequency} \rangle$ pairs. The map function emits a $\langle \text{hostname}, \text{term vector} \rangle$ pair for each input document (where the hostname is extracted from the URL of the document). The reduce function is passed all per-document term vectors for a given host. It adds these term vectors together, throwing away infrequent terms, and then emits a final $\langle \text{hostname}, \text{term vector} \rangle$ pair.

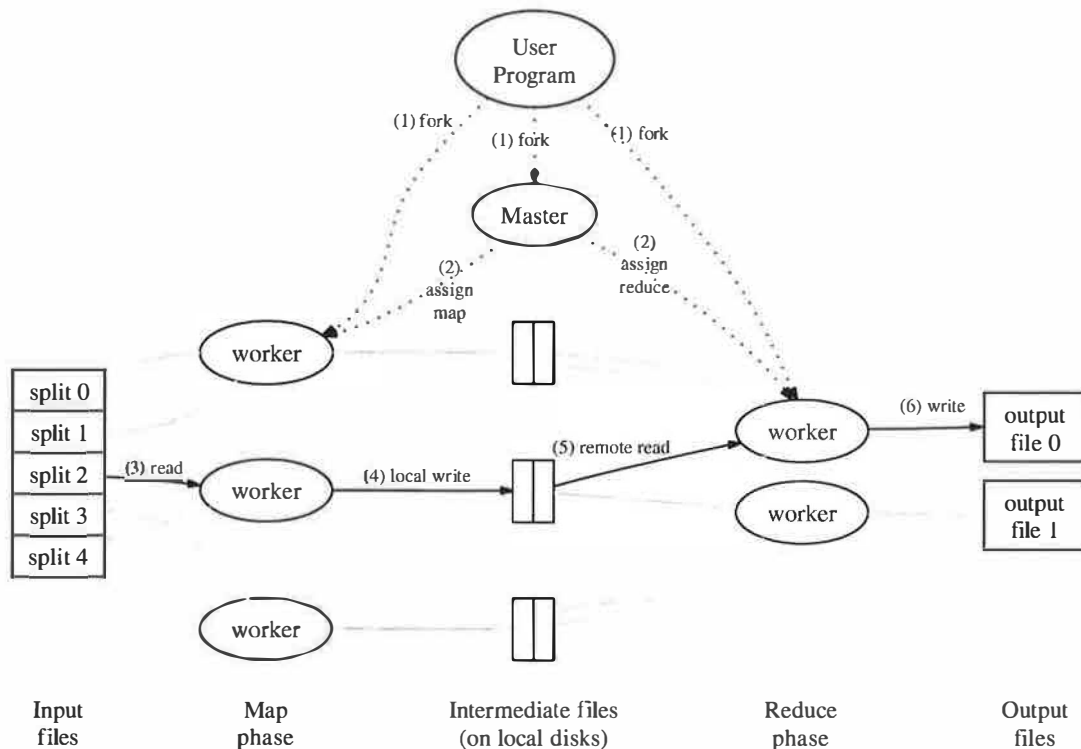


Figure 1: Execution overview

Inverted Index: The map function parses each document, and emits a sequence of $\langle \text{word}, \text{document ID} \rangle$ pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a $\langle \text{word}, \text{list}(\text{document ID}) \rangle$ pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

Distributed Sort: The map function extracts the key from each record, and emits a $\langle \text{key}, \text{record} \rangle$ pair. The reduce function emits all pairs unchanged. This computation depends on the partitioning facilities described in Section 4.1 and the ordering properties described in Section 4.2.

3 Implementation

Many different implementations of the MapReduce interface are possible. The right choice depends on the environment. For example, one implementation may be suitable for a small shared-memory machine, another for a large NUMA multi-processor, and yet another for an even larger collection of networked machines.

This section describes an implementation targeted to the computing environment in wide use at Google:

large clusters of commodity PCs connected together with switched Ethernet [4]. In our environment:

- (1) Machines are typically dual-processor x86 processors running Linux, with 2-4 GB of memory per machine.
- (2) Commodity networking hardware is used – typically either 100 megabits/second or 1 gigabit/second at the machine level, but averaging considerably less in overall bisection bandwidth.
- (3) A cluster consists of hundreds or thousands of machines, and therefore machine failures are common.
- (4) Storage is provided by inexpensive IDE disks attached directly to individual machines. A distributed file system [8] developed in-house is used to manage the data stored on these disks. The file system uses replication to provide availability and reliability on top of unreliable hardware.
- (5) Users submit jobs to a scheduling system. Each job consists of a set of tasks, and is mapped by the scheduler to a set of available machines within a cluster.

3.1 Execution Overview

The *Map* invocations are distributed across multiple machines by automatically partitioning the input data

into a set of M splits. The input splits can be processed in parallel by different machines. *Reduce* invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g., $\text{hash}(\text{key}) \bmod R$). The number of partitions (R) and the partitioning function are specified by the user.

Figure 1 shows the overall flow of a MapReduce operation in our implementation. When the user program calls the MapReduce function, the following sequence of actions occurs (the numbered labels in Figure 1 correspond to the numbers in the list below):

1. The MapReduce library in the user program first splits the input files into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece (controllable by the user via an optional parameter). It then starts up many copies of the program on a cluster of machines.
2. One of the copies of the program is special – the master. The rest are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.
3. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user-defined *Map* function. The intermediate key/value pairs produced by the *Map* function are buffered in memory.
4. Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.
5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used.
6. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's *Reduce* function. The output of the *Reduce* function is appended to a final output file for this reduce partition.

7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the MapReduce call in the user program returns back to the user code.

After successful completion, the output of the mapreduce execution is available in the R output files (one per reduce task, with file names as specified by the user). Typically, users do not need to combine these R output files into one file – they often pass these files as input to another MapReduce call, or use them from another distributed application that is able to deal with input that is partitioned into multiple files.

3.2 Master Data Structures

The master keeps several data structures. For each map task and reduce task, it stores the state (*idle*, *in-progress*, or *completed*), and the identity of the worker machine (for non-idle tasks).

The master is the conduit through which the location of intermediate file regions is propagated from map tasks to reduce tasks. Therefore, for each completed map task, the master stores the locations and sizes of the R intermediate file regions produced by the map task. Updates to this location and size information are received as map tasks are completed. The information is pushed incrementally to workers that have *in-progress* reduce tasks.

3.3 Fault Tolerance

Since the MapReduce library is designed to help process very large amounts of data using hundreds or thousands of machines, the library must tolerate machine failures gracefully.

Worker Failure

The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed. Any map tasks completed by the worker are reset back to their initial *idle* state, and therefore become eligible for scheduling on other workers. Similarly, any map task or reduce task in progress on a failed worker is also reset to *idle* and becomes eligible for rescheduling.

Completed map tasks are re-executed on a failure because their output is stored on the local disk(s) of the failed machine and is therefore inaccessible. Completed reduce tasks do not need to be re-executed since their output is stored in a global file system.

When a map task is executed first by worker A and then later executed by worker B (because A failed), all

workers executing reduce tasks are notified of the re-execution. Any reduce task that has not already read the data from worker *A* will read the data from worker *B*.

MapReduce is resilient to large-scale worker failures. For example, during one MapReduce operation, network maintenance on a running cluster was causing groups of 80 machines at a time to become unreachable for several minutes. The MapReduce master simply re-executed the work done by the unreachable worker machines, and continued to make forward progress, eventually completing the MapReduce operation.

Master Failure

It is easy to make the master write periodic checkpoints of the master data structures described above. If the master task dies, a new copy can be started from the last checkpointed state. However, given that there is only a single master, its failure is unlikely; therefore our current implementation aborts the MapReduce computation if the master fails. Clients can check for this condition and retry the MapReduce operation if they desire.

Semantics in the Presence of Failures

When the user-supplied *map* and *reduce* operators are deterministic functions of their input values, our distributed implementation produces the same output as would have been produced by a non-faulting sequential execution of the entire program.

We rely on atomic commits of map and reduce task outputs to achieve this property. Each in-progress task writes its output to private temporary files. A reduce task produces one such file, and a map task produces *R* such files (one per reduce task). When a map task completes, the worker sends a message to the master and includes the names of the *R* temporary files in the message. If the master receives a completion message for an already completed map task, it ignores the message. Otherwise, it records the names of *R* files in a master data structure.

When a reduce task completes, the reduce worker atomically renames its temporary output file to the final output file. If the same reduce task is executed on multiple machines, multiple rename calls will be executed for the same final output file. We rely on the atomic rename operation provided by the underlying file system to guarantee that the final file system state contains just the data produced by one execution of the reduce task.

The vast majority of our *map* and *reduce* operators are deterministic, and the fact that our semantics are equivalent to a sequential execution in this case makes it very

easy for programmers to reason about their program's behavior. When the *map* and/or *reduce* operators are non-deterministic, we provide weaker but still reasonable semantics. In the presence of non-deterministic operators, the output of a particular reduce task *R*₁ is equivalent to the output for *R*₁ produced by a sequential execution of the non-deterministic program. However, the output for a different reduce task *R*₂ may correspond to the output for *R*₂ produced by a different sequential execution of the non-deterministic program.

Consider map task *M* and reduce tasks *R*₁ and *R*₂. Let *e*(*R*_{*i*}) be the execution of *R*_{*i*} that committed (there is exactly one such execution). The weaker semantics arise because *e*(*R*₁) may have read the output produced by one execution of *M* and *e*(*R*₂) may have read the output produced by a different execution of *M*.

3.4 Locality

Network bandwidth is a relatively scarce resource in our computing environment. We conserve network bandwidth by taking advantage of the fact that the input data (managed by GFS [8]) is stored on the local disks of the machines that make up our cluster. GFS divides each file into 64 MB blocks, and stores several copies of each block (typically 3 copies) on different machines. The MapReduce master takes the location information of the input files into account and attempts to schedule a map task on a machine that contains a replica of the corresponding input data. Failing that, it attempts to schedule a map task near a replica of that task's input data (e.g., on a worker machine that is on the same network switch as the machine containing the data). When running large MapReduce operations on a significant fraction of the workers in a cluster, most input data is read locally and consumes no network bandwidth.

3.5 Task Granularity

We subdivide the map phase into *M* pieces and the reduce phase into *R* pieces, as described above. Ideally, *M* and *R* should be much larger than the number of worker machines. Having each worker perform many different tasks improves dynamic load balancing, and also speeds up recovery when a worker fails: the many map tasks it has completed can be spread out across all the other worker machines.

There are practical bounds on how large *M* and *R* can be in our implementation, since the master must make *O*(*M* + *R*) scheduling decisions and keeps *O*(*M* * *R*) state in memory as described above. (The constant factors for memory usage are small however: the *O*(*M* * *R*) piece of the state consists of approximately one byte of data per map task/reduce task pair.)

Furthermore, R is often constrained by users because the output of each reduce task ends up in a separate output file. In practice, we tend to choose M so that each individual task is roughly 16 MB to 64 MB of input data (so that the locality optimization described above is most effective), and we make R a small multiple of the number of worker machines we expect to use. We often perform MapReduce computations with $M = 200,000$ and $R = 5,000$, using 2,000 worker machines.

3.6 Backup Tasks

One of the common causes that lengthens the total time taken for a MapReduce operation is a “straggler”: a machine that takes an unusually long time to complete one of the last few map or reduce tasks in the computation. Stragglers can arise for a whole host of reasons. For example, a machine with a bad disk may experience frequent correctable errors that slow its read performance from 30 MB/s to 1 MB/s. The cluster scheduling system may have scheduled other tasks on the machine, causing it to execute the MapReduce code more slowly due to competition for CPU, memory, local disk, or network bandwidth. A recent problem we experienced was a bug in machine initialization code that caused processor caches to be disabled: computations on affected machines slowed down by over a factor of one hundred.

We have a general mechanism to alleviate the problem of stragglers. When a MapReduce operation is close to completion, the master schedules backup executions of the remaining *in-progress* tasks. The task is marked as completed whenever either the primary or the backup execution completes. We have tuned this mechanism so that it typically increases the computational resources used by the operation by no more than a few percent. We have found that this significantly reduces the time to complete large MapReduce operations. As an example, the sort program described in Section 5.3 takes 44% longer to complete when the backup task mechanism is disabled.

4 Refinements

Although the basic functionality provided by simply writing *Map* and *Reduce* functions is sufficient for most needs, we have found a few extensions useful. These are described in this section.

4.1 Partitioning Function

The users of MapReduce specify the number of reduce tasks/output files that they desire (R). Data gets partitioned across these tasks using a partitioning function on

the intermediate key. A default partitioning function is provided that uses hashing (e.g. “ $\text{hash}(\text{key}) \bmod R$ ”). This tends to result in fairly well-balanced partitions. In some cases, however, it is useful to partition data by some other function of the key. For example, sometimes the output keys are URLs, and we want all entries for a single host to end up in the same output file. To support situations like this, the user of the MapReduce library can provide a special partitioning function. For example, using “ $\text{hash}(\text{Hostname}(\text{urlkey})) \bmod R$ ” as the partitioning function causes all URLs from the same host to end up in the same output file.

4.2 Ordering Guarantees

We guarantee that within a given partition, the intermediate key/value pairs are processed in increasing key order. This ordering guarantee makes it easy to generate a sorted output file per partition, which is useful when the output file format needs to support efficient random access lookups by key, or users of the output find it convenient to have the data sorted.

4.3 Combiner Function

In some cases, there is significant repetition in the intermediate keys produced by each map task, and the user-specified *Reduce* function is commutative and associative. A good example of this is the word counting example in Section 2.1. Since word frequencies tend to follow a Zipf distribution, each map task will produce hundreds or thousands of records of the form $\langle \text{the}, 1 \rangle$. All of these counts will be sent over the network to a single reduce task and then added together by the *Reduce* function to produce one number. We allow the user to specify an optional *Combiner* function that does partial merging of this data before it is sent over the network.

The *Combiner* function is executed on each machine that performs a map task. Typically the same code is used to implement both the combiner and the reduce functions. The only difference between a reduce function and a combiner function is how the MapReduce library handles the output of the function. The output of a reduce function is written to the final output file. The output of a combiner function is written to an intermediate file that will be sent to a reduce task.

Partial combining significantly speeds up certain classes of MapReduce operations. Appendix A contains an example that uses a combiner.

4.4 Input and Output Types

The MapReduce library provides support for reading input data in several different formats. For example, “text”

mode input treats each line as a key/value pair: the key is the offset in the file and the value is the contents of the line. Another common supported format stores a sequence of key/value pairs sorted by key. Each input type implementation knows how to split itself into meaningful ranges for processing as separate map tasks (e.g. text mode's range splitting ensures that range splits occur only at line boundaries). Users can add support for a new input type by providing an implementation of a simple *reader* interface, though most users just use one of a small number of predefined input types.

A *reader* does not necessarily need to provide data read from a file. For example, it is easy to define a *reader* that reads records from a database, or from data structures mapped in memory.

In a similar fashion, we support a set of output types for producing data in different formats and it is easy for user code to add support for new output types.

4.5 Side-effects

In some cases, users of MapReduce have found it convenient to produce auxiliary files as additional outputs from their map and/or reduce operators. We rely on the application writer to make such side-effects atomic and idempotent. Typically the application writes to a temporary file and atomically renames this file once it has been fully generated.

We do not provide support for atomic two-phase commits of multiple output files produced by a single task. Therefore, tasks that produce multiple output files with cross-file consistency requirements should be deterministic. This restriction has never been an issue in practice.

4.6 Skipping Bad Records

Sometimes there are bugs in user code that cause the *Map* or *Reduce* functions to crash deterministically on certain records. Such bugs prevent a MapReduce operation from completing. The usual course of action is to fix the bug, but sometimes this is not feasible; perhaps the bug is in a third-party library for which source code is unavailable. Also, sometimes it is acceptable to ignore a few records, for example when doing statistical analysis on a large data set. We provide an optional mode of execution where the MapReduce library detects which records cause deterministic crashes and skips these records in order to make forward progress.

Each worker process installs a signal handler that catches segmentation violations and bus errors. Before invoking a user *Map* or *Reduce* operation, the MapReduce library stores the sequence number of the argument in a global variable. If the user code generates a signal,

the signal handler sends a "last gasp" UDP packet that contains the sequence number to the MapReduce master. When the master has seen more than one failure on a particular record, it indicates that the record should be skipped when it issues the next re-execution of the corresponding Map or Reduce task.

4.7 Local Execution

Debugging problems in *Map* or *Reduce* functions can be tricky, since the actual computation happens in a distributed system, often on several thousand machines, with work assignment decisions made dynamically by the master. To help facilitate debugging, profiling, and small-scale testing, we have developed an alternative implementation of the MapReduce library that sequentially executes all of the work for a MapReduce operation on the local machine. Controls are provided to the user so that the computation can be limited to particular map tasks. Users invoke their program with a special flag and can then easily use any debugging or testing tools they find useful (e.g. *gdb*).

4.8 Status Information

The master runs an internal HTTP server and exports a set of status pages for human consumption. The status pages show the progress of the computation, such as how many tasks have been completed, how many are in progress, bytes of input, bytes of intermediate data, bytes of output, processing rates, etc. The pages also contain links to the standard error and standard output files generated by each task. The user can use this data to predict how long the computation will take, and whether or not more resources should be added to the computation. These pages can also be used to figure out when the computation is much slower than expected.

In addition, the top-level status page shows which workers have failed, and which map and reduce tasks they were processing when they failed. This information is useful when attempting to diagnose bugs in the user code.

4.9 Counters

The MapReduce library provides a counter facility to count occurrences of various events. For example, user code may want to count total number of words processed or the number of German documents indexed, etc.

To use this facility, user code creates a named counter object and then increments the counter appropriately in the *Map* and/or *Reduce* function. For example:

```

Counter* uppercase;
uppercase = GetCounter("uppercase");

map(String name, String contents):
  for each word w in contents:
    if (IsCapitalized(w)):
      uppercase->Increment();
      EmitIntermediate(w, "1");

```

The counter values from individual worker machines are periodically propagated to the master (piggybacked on the ping response). The master aggregates the counter values from successful map and reduce tasks and returns them to the user code when the MapReduce operation is completed. The current counter values are also displayed on the master status page so that a human can watch the progress of the live computation. When aggregating counter values, the master eliminates the effects of duplicate executions of the same map or reduce task to avoid double counting. (Duplicate executions can arise from our use of backup tasks and from re-execution of tasks due to failures.)

Some counter values are automatically maintained by the MapReduce library, such as the number of input key/value pairs processed and the number of output key/value pairs produced.

Users have found the counter facility useful for sanity checking the behavior of MapReduce operations. For example, in some MapReduce operations, the user code may want to ensure that the number of output pairs produced exactly equals the number of input pairs processed, or that the fraction of German documents processed is within some tolerable fraction of the total number of documents processed.

5 Performance

In this section we measure the performance of MapReduce on two computations running on a large cluster of machines. One computation searches through approximately one terabyte of data looking for a particular pattern. The other computation sorts approximately one terabyte of data.

These two programs are representative of a large subset of the real programs written by users of MapReduce – one class of programs shuffles data from one representation to another, and another class extracts a small amount of interesting data from a large data set.

5.1 Cluster Configuration

All of the programs were executed on a cluster that consisted of approximately 1800 machines. Each machine had two 2GHz Intel Xeon processors with Hyper-Threading enabled, 4GB of memory, two 160GB IDE

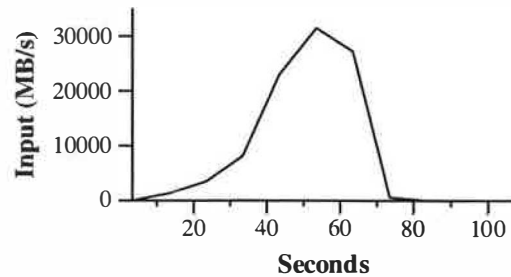


Figure 2: Data transfer rate over time

disks, and a gigabit Ethernet link. The machines were arranged in a two-level tree-shaped switched network with approximately 100-200 Gbps of aggregate bandwidth available at the root. All of the machines were in the same hosting facility and therefore the round-trip time between any pair of machines was less than a millisecond.

Out of the 4GB of memory, approximately 1-1.5GB was reserved by other tasks running on the cluster. The programs were executed on a weekend afternoon, when the CPUs, disks, and network were mostly idle.

5.2 Grep

The *grep* program scans through 10^{10} 100-byte records, searching for a relatively rare three-character pattern (the pattern occurs in 92,337 records). The input is split into approximately 64MB pieces ($M = 15000$), and the entire output is placed in one file ($R = 1$).

Figure 2 shows the progress of the computation over time. The Y-axis shows the rate at which the input data is scanned. The rate gradually picks up as more machines are assigned to this MapReduce computation, and peaks at over 30 GB/s when 1764 workers have been assigned. As the map tasks finish, the rate starts dropping and hits zero about 80 seconds into the computation. The entire computation takes approximately 150 seconds from start to finish. This includes about a minute of startup overhead. The overhead is due to the propagation of the program to all worker machines, and delays interacting with GFS to open the set of 1000 input files and to get the information needed for the locality optimization.

5.3 Sort

The *sort* program sorts 10^{10} 100-byte records (approximately 1 terabyte of data). This program is modeled after the TeraSort benchmark [10].

The sorting program consists of less than 50 lines of user code. A three-line *Map* function extracts a 10-byte sorting key from a text line and emits the key and the

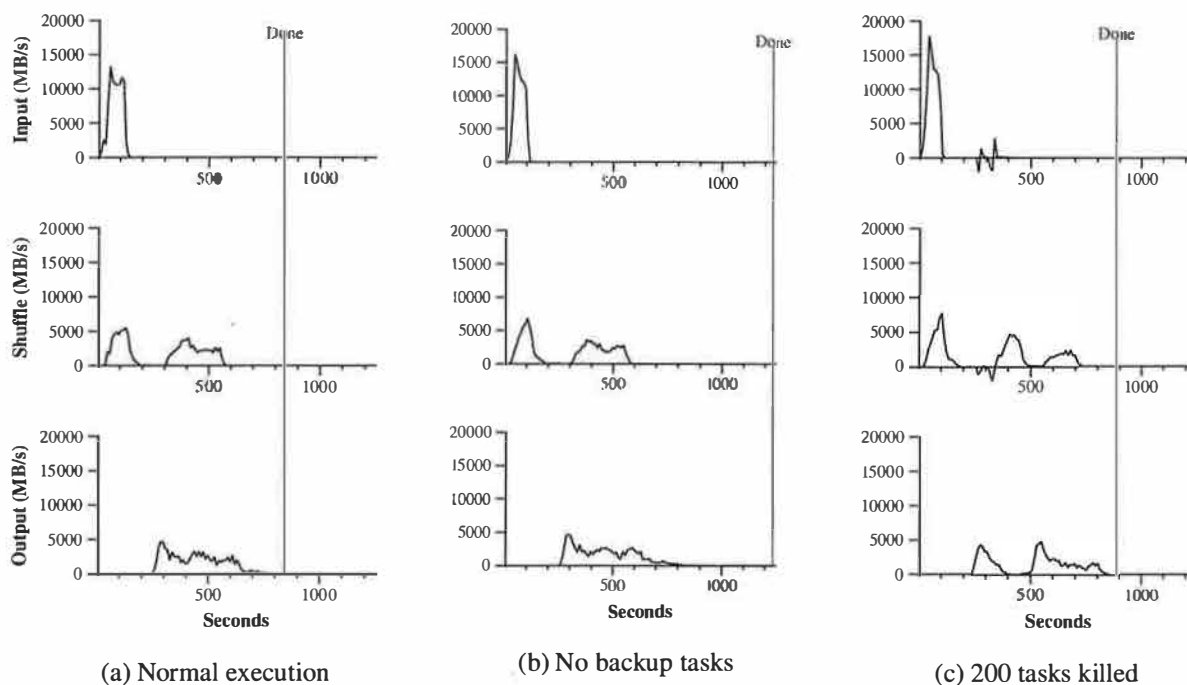


Figure 3: Data transfer rates over time for different executions of the sort program

original text line as the intermediate key/value pair. We used a built-in *Identity* function as the *Reduce* operator. This function passes the intermediate key/value pair unchanged as the output key/value pair. The final sorted output is written to a set of 2-way replicated GFS files (i.e., 2 terabytes are written as the output of the program).

As before, the input data is split into 64MB pieces ($M = 15000$). We partition the sorted output into 4000 files ($R = 4000$). The partitioning function uses the initial bytes of the key to segregate it into one of R pieces.

Our partitioning function for this benchmark has built-in knowledge of the distribution of keys. In a general sorting program, we would add a pre-pass MapReduce operation that would collect a sample of the keys and use the distribution of the sampled keys to compute split-points for the final sorting pass.

Figure 3 (a) shows the progress of a normal execution of the sort program. The top-left graph shows the rate at which input is read. The rate peaks at about 13 GB/s and dies off fairly quickly since all map tasks finish before 200 seconds have elapsed. Note that the input rate is less than for *grep*. This is because the sort map tasks spend about half their time and I/O bandwidth writing intermediate output to their local disks. The corresponding intermediate output for *grep* had negligible size.

The middle-left graph shows the rate at which data is sent over the network from the map tasks to the reduce tasks. This shuffling starts as soon as the first map task completes. The first hump in the graph is for

the first batch of approximately 1700 reduce tasks (the entire MapReduce was assigned about 1700 machines, and each machine executes at most one reduce task at a time). Roughly 300 seconds into the computation, some of these first batch of reduce tasks finish and we start shuffling data for the remaining reduce tasks. All of the shuffling is done about 600 seconds into the computation.

The bottom-left graph shows the rate at which sorted data is written to the final output files by the reduce tasks. There is a delay between the end of the first shuffling period and the start of the writing period because the machines are busy sorting the intermediate data. The writes continue at a rate of about 2-4 GB/s for a while. All of the writes finish about 850 seconds into the computation. Including startup overhead, the entire computation takes 891 seconds. This is similar to the current best reported result of 1057 seconds for the TeraSort benchmark [18].

A few things to note: the input rate is higher than the shuffle rate and the output rate because of our locality optimization – most data is read from a local disk and bypasses our relatively bandwidth constrained network. The shuffle rate is higher than the output rate because the output phase writes two copies of the sorted data (we make two replicas of the output for reliability and availability reasons). We write two replicas because that is the mechanism for reliability and availability provided by our underlying file system. Network bandwidth requirements for writing data would be reduced if the underlying file system used erasure coding [14] rather than replication.

5.4 Effect of Backup Tasks

In Figure 3 (b), we show an execution of the sort program with backup tasks disabled. The execution flow is similar to that shown in Figure 3 (a), except that there is a very long tail where hardly any write activity occurs. After 960 seconds, all except 5 of the reduce tasks are completed. However these last few stragglers don't finish until 300 seconds later. The entire computation takes 1283 seconds, an increase of 44% in elapsed time.

5.5 Machine Failures

In Figure 3 (c), we show an execution of the sort program where we intentionally killed 200 out of 1746 worker processes several minutes into the computation. The underlying cluster scheduler immediately restarted new worker processes on these machines (since only the processes were killed, the machines were still functioning properly).

The worker deaths show up as a negative input rate since some previously completed map work disappears (since the corresponding map workers were killed) and needs to be redone. The re-execution of this map work happens relatively quickly. The entire computation finishes in 933 seconds including startup overhead (just an increase of 5% over the normal execution time).

6 Experience

We wrote the first version of the MapReduce library in February of 2003, and made significant enhancements to it in August of 2003, including the locality optimization, dynamic load balancing of task execution across worker machines, etc. Since that time, we have been pleasantly surprised at how broadly applicable the MapReduce library has been for the kinds of problems we work on. It has been used across a wide range of domains within Google, including:

- large-scale machine learning problems,
- clustering problems for the Google News and Froogle products,
- extraction of data used to produce reports of popular queries (e.g. Google Zeitgeist),
- extraction of properties of web pages for new experiments and products (e.g. extraction of geographical locations from a large corpus of web pages for localized search), and
- large-scale graph computations.

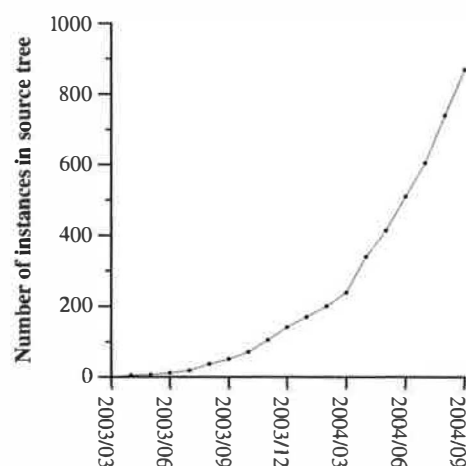


Figure 4: MapReduce instances over time

Number of jobs	29,423
Average job completion time	634 secs
Machine days used	79,186 days
Input data read	3,288 TB
Intermediate data produced	758 TB
Output data written	193 TB
Average worker machines per job	157
Average worker deaths per job	1.2
Average map tasks per job	3,351
Average reduce tasks per job	55
Unique <i>map</i> implementations	395
Unique <i>reduce</i> implementations	269
Unique <i>map/reduce</i> combinations	426

Table 1: MapReduce jobs run in August 2004

Figure 4 shows the significant growth in the number of separate MapReduce programs checked into our primary source code management system over time, from 0 in early 2003 to almost 900 separate instances as of late September 2004. MapReduce has been so successful because it makes it possible to write a simple program and run it efficiently on a thousand machines in the course of half an hour, greatly speeding up the development and prototyping cycle. Furthermore, it allows programmers who have no experience with distributed and/or parallel systems to exploit large amounts of resources easily.

At the end of each job, the MapReduce library logs statistics about the computational resources used by the job. In Table 1, we show some statistics for a subset of MapReduce jobs run at Google in August 2004.

6.1 Large-Scale Indexing

One of our most significant uses of MapReduce to date has been a complete rewrite of the production index-

ing system that produces the data structures used for the Google web search service. The indexing system takes as input a large set of documents that have been retrieved by our crawling system, stored as a set of GFS files. The raw contents for these documents are more than 20 terabytes of data. The indexing process runs as a sequence of five to ten MapReduce operations. Using MapReduce (instead of the ad-hoc distributed passes in the prior version of the indexing system) has provided several benefits:

- The indexing code is simpler, smaller, and easier to understand, because the code that deals with fault tolerance, distribution and parallelization is hidden within the MapReduce library. For example, the size of one phase of the computation dropped from approximately 3800 lines of C++ code to approximately 700 lines when expressed using MapReduce.
- The performance of the MapReduce library is good enough that we can keep conceptually unrelated computations separate, instead of mixing them together to avoid extra passes over the data. This makes it easy to change the indexing process. For example, one change that took a few months to make in our old indexing system took only a few days to implement in the new system.
- The indexing process has become much easier to operate, because most of the problems caused by machine failures, slow machines, and networking hiccups are dealt with automatically by the MapReduce library without operator intervention. Furthermore, it is easy to improve the performance of the indexing process by adding new machines to the indexing cluster.

7 Related Work

Many systems have provided restricted programming models and used the restrictions to parallelize the computation automatically. For example, an associative function can be computed over all prefixes of an N element array in $\log N$ time on N processors using parallel prefix computations [6, 9, 13]. MapReduce can be considered a simplification and distillation of some of these models based on our experience with large real-world computations. More significantly, we provide a fault-tolerant implementation that scales to thousands of processors. In contrast, most of the parallel processing systems have only been implemented on smaller scales and leave the details of handling machine failures to the programmer.

Bulk Synchronous Programming [17] and some MPI primitives [11] provide higher-level abstractions that

make it easier for programmers to write parallel programs. A key difference between these systems and MapReduce is that MapReduce exploits a restricted programming model to parallelize the user program automatically and to provide transparent fault-tolerance.

Our locality optimization draws its inspiration from techniques such as active disks [12, 15], where computation is pushed into processing elements that are close to local disks, to reduce the amount of data sent across I/O subsystems or the network. We run on commodity processors to which a small number of disks are directly connected instead of running directly on disk controller processors, but the general approach is similar.

Our backup task mechanism is similar to the eager scheduling mechanism employed in the Charlotte System [3]. One of the shortcomings of simple eager scheduling is that if a given task causes repeated failures, the entire computation fails to complete. We fix some instances of this problem with our mechanism for skipping bad records.

The MapReduce implementation relies on an in-house cluster management system that is responsible for distributing and running user tasks on a large collection of shared machines. Though not the focus of this paper, the cluster management system is similar in spirit to other systems such as Condor [16].

The sorting facility that is a part of the MapReduce library is similar in operation to NOW-Sort [1]. Source machines (map workers) partition the data to be sorted and send it to one of R reduce workers. Each reduce worker sorts its data locally (in memory if possible). Of course NOW-Sort does not have the user-definable Map and Reduce functions that make our library widely applicable.

River [2] provides a programming model where processes communicate with each other by sending data over distributed queues. Like MapReduce, the River system tries to provide good average case performance even in the presence of non-uniformities introduced by heterogeneous hardware or system perturbations. River achieves this by careful scheduling of disk and network transfers to achieve balanced completion times. MapReduce has a different approach. By restricting the programming model, the MapReduce framework is able to partition the problem into a large number of fine-grained tasks. These tasks are dynamically scheduled on available workers so that faster workers process more tasks. The restricted programming model also allows us to schedule redundant executions of tasks near the end of the job which greatly reduces completion time in the presence of non-uniformities (such as slow or stuck workers).

BAD-FS [5] has a very different programming model from MapReduce, and unlike MapReduce, is targeted to

the execution of jobs across a wide-area network. However, there are two fundamental similarities. (1) Both systems use redundant execution to recover from data loss caused by failures. (2) Both use locality-aware scheduling to reduce the amount of data sent across congested network links.

TACC [7] is a system designed to simplify construction of highly-available networked services. Like MapReduce, it relies on re-execution as a mechanism for implementing fault-tolerance.

8 Conclusions

The MapReduce programming model has been successfully used at Google for many different purposes. We attribute this success to several reasons. First, the model is easy to use, even for programmers without experience with parallel and distributed systems, since it hides the details of parallelization, fault-tolerance, locality optimization, and load balancing. Second, a large variety of problems are easily expressible as MapReduce computations. For example, MapReduce is used for the generation of data for Google's production web search service, for sorting, for data mining, for machine learning, and many other systems. Third, we have developed an implementation of MapReduce that scales to large clusters of machines comprising thousands of machines. The implementation makes efficient use of these machine resources and therefore is suitable for use on many of the large computational problems encountered at Google.

We have learned several things from this work. First, restricting the programming model makes it easy to parallelize and distribute computations and to make such computations fault-tolerant. Second, network bandwidth is a scarce resource. A number of optimizations in our system are therefore targeted at reducing the amount of data sent across the network: the locality optimization allows us to read data from local disks, and writing a single copy of the intermediate data to local disk saves network bandwidth. Third, redundant execution can be used to reduce the impact of slow machines, and to handle machine failures and data loss.

Acknowledgements

Josh Levenberg has been instrumental in revising and extending the user-level MapReduce API with a number of new features based on his experience with using MapReduce and other people's suggestions for enhancements. MapReduce reads its input from and writes its output to the Google File System [8]. We would like to thank Mohit Aron, Howard Gobioff, Markus Gutschke,

David Kramer, Shun-Tak Leung, and Josh Redstone for their work in developing GFS. We would also like to thank Percy Liang and Olcan Sercinoglu for their work in developing the cluster management system used by MapReduce. Mike Burrows, Wilson Hsieh, Josh Levenberg, Sharon Perl, Rob Pike, and Debby Wallach provided helpful comments on earlier drafts of this paper. The anonymous OSDI reviewers, and our shepherd, Eric Brewer, provided many useful suggestions of areas where the paper could be improved. Finally, we thank all the users of MapReduce within Google's engineering organization for providing helpful feedback, suggestions, and bug reports.

References

- [1] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-performance sorting on networks of workstations. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 1997.
- [2] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS '99)*, pages 10–22, Atlanta, Georgia, May 1999.
- [3] Arash Baratloo, Mehmet Karaul, Zvi Kedem, and Peter Wyckoff. Charlotte: Metacomputing on the web. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems*, 1996.
- [4] Luiz A. Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, April 2003.
- [5] John Bent, Douglas Thain, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Explicit control in a batch-aware distributed file system. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation NSDI*, March 2004.
- [6] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C-38(11), November 1989.
- [7] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 78–91, Saint-Malo, France, 1997.
- [8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *19th Symposium on Operating Systems Principles*, pages 29–43, Lake George, New York, 2003.

- [9] S. Gortatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96. Parallel Processing*, Lecture Notes in Computer Science 1124, pages 401–408. Springer-Verlag, 1996.
- [10] Jim Gray. Sort benchmark home page. <http://research.microsoft.com/barc/SortBenchmark/>.
- [11] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.
- [12] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *Proceedings of the 2004 USENIX File and Storage Technologies FAST Conference*, April 2004.
- [13] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.
- [14] Michael O. Rabin. Efficient dispersal of information for security, load balancing and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.
- [15] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. *IEEE Computer*, pages 68–74, June 2001.
- [16] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The Condor experience. *Concurrency and Computation: Practice and Experience*, 2004.
- [17] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1997.
- [18] Jim Wyllie. Spsort: How to sort a terabyte quickly. <http://almel.almaden.ibm.com/cs/spsort.pdf>.

A Word Frequency

This section contains a program that counts the number of occurrences of each unique word in a set of input files specified on the command line.

```
#include "mapreduce/mapreduce.h"

// User's map function
class WordCounter : public Mapper {
public:
    virtual void Map(const MapInput& input) {
        const string& text = input.value();
        const int n = text.size();
        for (int i = 0; i < n; ) {
            // Skip past leading whitespace
            while ((i < n) && isspace(text[i]))
                i++;

            // Find word end
            int start = i;
            while ((i < n) && !isspace(text[i]))
                i++;
```

```
            if (start < i)
                Emit(text.substr(start, i-start), "1");
        }
    };
REGISTER_MAPPER(WordCounter);

// User's reduce function
class Adder : public Reducer {
    virtual void Reduce(ReduceInput* input) {
        // Iterate over all entries with the
        // same key and add the values
        int64 value = 0;
        while (!input->done()) {
            value += StringToInt(input->value());
            input->NextValue();
        }

        // Emit sum for input->key()
        Emit(IntToString(value));
    }
};
REGISTER_REDUCER(Adder);

int main(int argc, char** argv) {
    ParseCommandLineFlags(argc, argv);

    MapReduceSpecification spec;

    // Store list of input files into "spec"
    for (int i = 1; i < argc; i++) {
        MapReduceInput* input = spec.add_input();
        input->set_format("text");
        input->set_filepattern(argv[i]);
        input->set_mapper_class("WordCounter");
    }

    // Specify the output files:
    // /gfs/test/freq-00000-of-00100
    // /gfs/test/freq-00001-of-00100
    // ...
    MapReduceOutput* out = spec.output();
    out->set_filebase("/gfs/test/freq");
    out->set_num_tasks(100);
    out->set_format("text");
    out->set_reducer_class("Adder");

    // Optional: do partial sums within map
    // tasks to save network bandwidth
    out->set_combiner_class("Adder");

    // Tuning parameters: use at most 2000
    // machines and 100 MB of memory per task
    spec.set_machines(2000);
    spec.set_map_megabytes(100);
    spec.set_reduce_megabytes(100);

    // Now run it
    MapReduceResult result;
    if (!MapReduce(spec, &result)) abort();

    // Done: 'result' structure contains info
    // about counters, time taken, number of
    // machines used, etc.

    return 0;
}
```


FUSE: Lightweight Guaranteed Distributed Failure Notification

John Dunagan*

Nicholas J. A. Harvey[†]
Marvin Theimer*

Michael B. Jones*
Alec Wolman*

Dejan Kostić[‡]

"I am not discouraged, because every failure is another step forward." – Thomas Edison

Abstract

FUSE is a lightweight failure notification service for building distributed systems. Distributed systems built with FUSE are guaranteed that failure notifications never fail. Whenever a failure notification is triggered, all live members of the FUSE group will hear a notification within a bounded period of time, irrespective of node or communication failures. In contrast to previous work on failure detection, the responsibility for deciding that a failure has occurred is shared between the FUSE service and the distributed application. This allows applications to implement their own definitions of failure. Our experience building a scalable distributed event delivery system on an overlay network has convinced us of the usefulness of this service. Our results demonstrate that the network costs of each FUSE group can be small; in particular, our overlay network implementation requires no additional liveness-verifying ping traffic beyond that already needed to maintain the overlay, making the steady state network load independent of the number of active FUSE groups.

1 Introduction

This paper describes FUSE, a lightweight failure notification service. When building distributed systems, managing failures is an important and often complex task. Many different architectures, abstractions, and services have been proposed to address this [5, 10, 13, 28, 30, 38, 42, 43, 44]. FUSE provides a new programming model for failure management that simplifies the task of agreeing when failures have occurred in a distributed system, thereby reducing the complexity faced by application developers. The most closely related prior work on coping with failures has centered around *failure detection services*. FUSE takes a somewhat different approach, where

detecting failures is a shared responsibility between FUSE and the application. Applications create a FUSE group with an immutable list of participants. FUSE monitors this group until either FUSE or the application decides to terminate the group, at which point all live participants are guaranteed to learn of a "group failure" within a bounded period of time. This focus on delivering failure notifications leads us to refer to FUSE as a *failure notification service*. This is a very different approach than prior systems have adopted, and we will argue that it is a good approach for wide-area Internet applications.

Applications make use of the FUSE abstraction as follows: the application asks FUSE to create a new group, specifying the other participating nodes. When FUSE finishes constructing the group, it returns a unique identifier for this group to the creator. The application then passes this FUSE ID to the applications on all the other nodes in this group, each of which registers a callback associated with the given FUSE ID. FUSE guarantees that every group member will be reliably notified via this callback whenever a failure condition affects the group. This failure notification may be triggered either explicitly, by the application, or implicitly, when FUSE detects that communication among group members is impaired.

Applications can create multiple FUSE groups for different purposes, even if those FUSE groups span the same set of nodes. In the event that FUSE detects a low-level communication failure, failure will be signalled on all the FUSE groups using that path. However, on any individual FUSE group the application may signal a failure without affecting any of the other groups.

FUSE provides the guarantee that notifications will be delivered within a bounded period of time, even in the face of node crashes and arbitrary network failures. We refer to these semantics as "distributed one-way agreement". One-way refers to the fact that there is only one transition any group member can see: from "live" to "failed". After failure notification on a group, detecting future failures requires creating a new group.

By providing these semantics, FUSE ensures that *failure notifications never fail*. This greatly simplifies failure handling among nodes that have state that they want to handle in a coordinated fashion. FUSE efficiently handles all the corner cases of guaranteeing that all members will be notified of any failure condition affecting the group. Applications built on top of FUSE do not need to worry

*Microsoft Research, Microsoft Corporation, Redmond, WA. {jdunagan, mbj, theimer, alecw}@microsoft.com

[†]Department of Computer Science, Duke University, Durham, NC. dkostic@cs.duke.edu

[‡]Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA. nickh@mit.edu

that a failure message did not get through or that orphaned state remains in the system.

The primary target for FUSE is wide-area Internet applications, such as content delivery networks, peer-to-peer applications, web services, and grid computing. FUSE is not targeted at applications that require strong consistency on replicated data, such as stock exchanges and missile control systems. Techniques such as virtual synchrony [6], Paxos [29], and BFT [10] have already proven to be effective in such environments. However, these techniques incur significant overhead and therefore have limited scalability. Furthermore, FUSE does not provide resilience to malicious participants, though it also does not preclude solving this problem at a higher layer.

Previous work on failure detection services uses “membership” as the fundamental abstraction: these services typically provide a list of each component in the system, and whether it is currently up or down. Membership services have seen widespread success as a building block for implementing higher level distributed services, such as consensus, and have even been deployed in such commercially important systems as the New York Stock Exchange [5]. However, one disadvantage of the membership abstraction is that it does not allow application components to have failed with respect to one action, but not with respect to another. For example, suppose a node is engaged in an operation with a peer and at some point fails to receive a timely response. The failure management service should support declaring that this operation has failed without requiring that a node or process be declared to have failed. The FUSE abstraction provides this flexibility: FUSE tracks whether individual application communication paths are currently working in a manner that is acceptable to the application.

The scenario described above is more likely to occur with wide-area Internet applications, which face a demanding operating environment: network congestion leads to variations in network loss rate and delay; intransitive connectivity failures (sometimes called partial connectivity failures) occur due to router or firewall misconfiguration [27, 31]; and individual network components such as links and routers can also fail. Under these conditions, it is difficult for applications to make good decisions based solely on the information provided by a membership service. A particular scenario illustrating why applications need to participate in deciding when failure has occurred is delivery of streaming content over the Internet: failure to achieve a certain threshold bandwidth may be unacceptable to such an application even though other applications are perfectly happy with the connectivity provided by that same network path.

Our FUSE implementation is scalable, where scaling is with respect to the number of groups: multiple failure notification groups can share liveness checking messages. Other implementations of the FUSE abstraction may sacrifice scalability in favor of increased security. Our FUSE implementation is designed to support large numbers of

small to moderate size groups. We do not attempt to efficiently support very large groups because we believe very large groups will tend to suffer from too-frequent failure notification, making them less useful.

Our FUSE implementation is particularly well-suited to applications using scalable overlay networks. Scalable overlay networks already do liveness checking to maintain their routing tables and FUSE can re-use this liveness checking traffic. In such a deployment, the network traffic required to implement FUSE is independent of the number of groups until a failure occurs; only creation and teardown of a group introduce a per-group overhead. FUSE can be implemented in the absence of a pre-existing overlay network – the implementation can construct its own overlay, or it can use an alternative liveness checking topology.

We have implemented FUSE on top of the SkipNet [25] scalable overlay network, and we have built a scalable event delivery application using FUSE. We evaluated our implementation using two main techniques: a discrete event simulator to evaluate its scalability, and a live system with 400 overlay participants (each running in its own process) running on a cluster of 40 workstations to evaluate correctness and performance. Both the simulator and the live system use an identical code base except for the base messaging layer. Our live system evaluation shows that our FUSE implementation is indeed lightweight; the latency of FUSE group creation is the latency of an RPC call to the furthest group member; the latency of explicit failure notification is similarly dominated by network latency; and we show that our implementation is robust to false positives caused by network packet loss.

In summary, the key contributions of this paper are:

- We present a novel abstraction, the FUSE failure notification group, that provides the semantics of distributed one-way agreement. These are desirable semantics in our chosen setting of wide-area Internet applications.
- We used FUSE in building a scalable event delivery service and we describe how it significantly reduced the complexity of this task.
- We implemented FUSE on top of a scalable overlay network. This allowed us to support FUSE without adding additional liveness checking. We experimentally evaluated the performance of our implementation on a live system with 400 virtual nodes.

2 Related Work

Failure detection has been the subject of more than two decades of research. This work can be broadly classified into unreliable failure detectors, weakly consistent membership services, and strongly consistent membership services. Unreliable failure detectors provide the weakest semantics directly, but they are a standard building block in constructing membership services with stronger semantics. Both weakly-consistent and strongly-consistent

membership services are based on the abstraction of a list of available or unavailable components (typically processes or machines). In contrast, a FUSE group ID is not bound to a process or machine and hence can be used in many contexts. For example, it can correspond to a set of several processes, or to related data stored on several different machines. This abstraction allows FUSE to provide novel semantics for distributed agreement, a subject we will elaborate on in our discussion of weakly consistent membership services.

Chandra et al. formalized the concept of unreliable failure detectors, and showed that one such detector was the weakest failure detector for solving consensus [11, 12]. Such detectors typically provide periodic heartbeating and callbacks saying whether the component is “responding” or “not responding.” These callbacks may be aggregate judgments based on sets of pings. Unreliable failure detectors provide the following semantic guarantee: fail-stop crashes will be identified as such within a bounded amount of time. There has been extensive work on such detectors, focusing on such aspects as interface design, scalability, rapidity of failure detection, and minimizing network load [1, 17, 21, 23, 38].

FUSE uses a similar lightweight mechanism (periodic heartbeats) to unreliable failure detectors, but provides stronger distributed agreement semantics. Unreliable failure detectors are typically used as a component in a membership service, and the membership service is responsible for implementing distributed agreement semantics.

Weakly consistent membership services have also been the subject of an extensive body of work [2, 19, 42, 44]. This work can be broadly classified as differing in speed of failure detection, accuracy (low rate of false positives), message load, and completeness (failed nodes are agreed to have failed by everyone in the system). Epidemic and gossip-style algorithms have been used to build highly scalable implementations of this service [19, 42]. Previous application areas that have been proposed for such membership services are distributed collaborative applications, online games, large-scale publish-subscribe systems, and multiple varieties of Web Services [5, 19]. These are the same application domains targeted by FUSE, and FUSE has similar overhead to a weakly consistent membership service in these settings. Typical characteristics of such applications are that many operations are idempotent or can be straightforwardly undone, operations can be re-attempted with a different set of participants, or the decision to retry can be deferred to the user (as in an instant messaging service).

One novel aspect of the FUSE abstraction is the ability to handle arbitrary network failures. In contrast, weakly consistent membership services provide semantic guarantees assuming only a fail-stop model. One kind of network failure where the FUSE abstraction is useful is an intransitive connectivity failure: A can reach B, B can reach C, but A cannot reach C. This class of network failures is hard for a weakly consistent membership service to

handle because the abstraction of a membership list limits the service to one of three choices, each of which has drawbacks:

- Declare one of the nodes experiencing the intransitive connectivity failure to have failed. This prevents the use of that node by any node that *can* reach it.
- Declare all of the nodes experiencing the intransitive failure to be alive because other nodes in the system can reach them. This may cause the application to block for the duration of the connectivity failure.
- Allow a persistent inconsistency among different nodes’ views of the membership list. This forces the application to deal with inconsistency explicitly, and therefore the membership service is no longer reducing the complexity burden on the application developer.

FUSE appropriately handles intransitive connectivity failures by allowing the application on a node experiencing a failure to declare the corresponding FUSE group to have failed. Other FUSE groups involving the same node but not utilizing a failed communication path can continue to operate. Application participation is required to achieve this because FUSE may not have detected the failure itself; like most weakly consistent membership services, FUSE typically monitors only a subset of all application-level communication paths.

FUSE would require application involvement in failure detection even if it monitored all communication paths. Consider a multi-tier service composed of a front-end, middle-tier, and back-end. Suppose the middle-tier component is available but misconfigured. FUSE allows the front-end to declare a failure, and to then perform appropriate failure recovery, such as finding another middle-tier from some pool of available machines. This difference in usage between membership services and FUSE reflects a difference in philosophy. Membership services try to proactively decide at a system level whether or not nodes and processes are available. FUSE provides a mechanism that applications can use to declare failures when application-level constraints (such as configuration) are violated.

Another contrast between the two approaches is that the FUSE abstraction enables “fate-sharing” among distributed data items. By associating these items with a single FUSE group, application developers can enforce that invalidating any one item will cause all the remaining data items to be invalidated. Weakly consistent membership services do not explicitly provide this tying together of distributed data.

Strongly consistent membership services share the abstraction of a membership list, but they also guarantee that all nodes in the system always see a consistent list through the use of atomic updates. Such membership services are an important component in building highly available and reliable systems using virtual synchrony. Notable examples of systems built using this approach are the New York

and Swiss Stock Exchanges, the French Air Traffic Control System, and the AEGIS Warship [2, 5, 6]. However, a limitation of virtual synchrony is that it has only been shown to perform well at small scales, such as five node systems [6].

Some network routing protocols, such as IS-IS [8], OSPF [33], and AutoNet [35], use mechanisms similar to FUSE. One similar aspect of AutoNet is its use of teardown and recreate to manage failures. Any link-state change causes all AutoNet switches to discard their link-state databases and rebuild the global routing table. OSPF and IS-IS take local link observations and propagate them throughout the network using link-state announcements. They tolerate arbitrary network failures using timers and explicit refreshes to maintain the link-state databases. FUSE also uses timers and keep-alives to tolerate arbitrary network failures. However, FUSE uses them to tie together sets of links that provide end-to-end connectivity between group members, and to provide an overall yes/no decision for whether connectivity is satisfactory.

A more distantly related area of prior work is black-box techniques for diagnosing failures. Such techniques use statistics or machine learning to distinguish successful and failed requests [9, 13, 14, 15, 16]. In contrast, FUSE assumes application developer participation and provides semantic guarantees to the developer. Another significant distinction is that black-box techniques typically require data aggregation in a central location for analysis; FUSE has no such requirement.

Distributed transactions are a well-known abstraction for simplifying distributed systems. Because FUSE provides weaker semantics than distributed transactions, FUSE can maintain its semantic guarantees under network failures that cause distributed transactions to block. Theoretical results on consensus show that the possibility of blocking is fundamental to any protocol for distributed transactions [22, 24].

Two of the design choices we made in building FUSE were also recommended by recent works dealing with the architectural design of network protocols. Ji et al. [26] surveyed hard-state and soft-state signaling mechanisms across a broad class of network protocols, and recommended a soft state approach combining timers with explicit revocation: FUSE does this. Mogul et al. [32] argued that state maintained by network protocol implementations should be exposed to clients of those protocols. As described in Section 6, we modified our overlay routing layer to expose a mechanism for FUSE to piggy-back content on overlay maintenance traffic.

3 FUSE Semantics and API

We begin by describing one simple approach to implementing the FUSE abstraction. Suppose every group member periodically pings every other group member with an “are you okay?” message. A group member that

is not okay for any reason, either because of node failure, network disconnect, network partition, or transient overload, will fail to respond to some ping. The member that initiated this missed ping will ensure that a failure notification propagates to the rest of the group by ceasing to respond itself to all pings for that FUSE group. Any individual observation of a failure is thus converted into a group failure notification. This mechanism allows failure notifications to be delivered despite any pattern of disconnections, partitions, or node failures. This specific FUSE implementation guarantees that failure notifications are propagated to every party within twice the periodic ping interval. Our implementation uses a different liveness checking topology, discussed in Section 5. It also uses a different ping retry policy; the retry policy and the guarantees on failure notification latency are discussed in Section 3.3.

The name FUSE is derived from the analogy to “laying a fuse” between the group members. Whenever any group member wishes to signal failure it can light the fuse, and this failure notification will propagate to all other group members as the fuse burns. A connectivity failure or node crash at any intermediate location along the fuse will cause the fuse to be lit there as well, and the fuse will then start burning in every direction away from the failure. This ensures that communication failures will not stop the progress of the failure notification. Also, once the fuse has burnt, it cannot be relit, analogous to how the FUSE facility only notifies the application once per FUSE group.

Many different FUSE implementations are possible. All implementations of the FUSE abstraction must provide distributed one-way agreement: failure notifications are delivered to all live group members under node crashes and arbitrary network failures. Different FUSE implementations may use different strategies for group creation, liveness checking topology, retry, programming interface, and persistence, with consequent variations in performance. In this section, we describe the choices that we made in our FUSE implementation, the resulting semantics that application developers will need to understand, and some of the alternative strategies that other FUSE implementations could use.

3.1 Programming Interface

We now present the FUSE API for our implementation. FUSE groups are created by calling *CreateGroup* with a desired set of member nodes. This generates a FUSE ID unique to this group, communicates it to the FUSE layers on all the specified members, and then returns the ID to the caller. Applications are subsequently expected to explicitly communicate the FUSE ID from the creator to the other group members. Applications learning about this FUSE ID register a handler for FUSE notifications using the *RegisterFailureHandler* function. In this design, the FUSE layer is not responsible for communicating the FUSE ID to applications on nodes other than the creator.

```

// Creates a FUSE notification group containing
// the nodes in the set
FuseId CreateGroup(NodeId[] set)

// Registers a callback function to be invoked
// when a notification occurs for the FUSE group
void RegisterFailureHandler
    (Callback handler, FuseId id)

// Allows the application to explicitly cause
// FUSE failure notification
void SignalFailure(FuseId id)

```

Figure 1. *The FUSE API*

We believe the most likely use of FUSE is to allow fate-sharing of distributed application state. Applications should learn about FUSE IDs with sufficient context to know what application state to associate with the FUSE ID. Though failure handlers can simply perform garbage collection of the associated application state, a handler is also free to attempt to re-establish the application state using a new FUSE group, or to execute arbitrary code. For brevity, we refer to all of these permissible application-level actions using the short-hand “garbage collection.”

The application handler is invoked whenever the FUSE layer believes a failure has occurred, either because of a node or communication failure or because the application explicitly signalled a failure event at one of the group members. If *RegisterFailureHandler* is called with a FUSE ID parameter that does not exist, perhaps because it has already been signalled, the handler callback is invoked immediately. Applications that wish to explicitly signal failure do so by calling the *SignalFailure* function.

3.2 Group Creation

Group creation can be implemented in one of two ways: it can return immediately, or it can block until all nodes in the group have been contacted. Returning immediately reduces latency, but because FUSE has not checked that all group members are alive, the application may perform expensive operations, only to have FUSE signal failure a short time later. In contrast, blocking until all members have been contacted increases creation latency, but decreases the likelihood that the FUSE group will immediately fail. We chose to implement blocking create; this provides application developers the semantic guarantee that if group creation returns successfully, all the group members were alive and reachable. A high enough rate of churn amongst group members could repeatedly prevent FUSE group creation from succeeding. However, based on the low latency of FUSE group creation (Section 7), such a high churn rate is likely to cause the system to fail in other ways before FUSE becomes a bottleneck.

When *CreateGroup* is called, FUSE generates a globally unique FUSE ID for the group. Each node is then contacted and asked to join the new FUSE group. If any group member is unreachable, all nodes that already learned of the new FUSE group are then notified of the failure. Group members that learned of the group but sub-

sequently become unreachable similarly detect the group failure through their inability to communicate with other group members. If the FUSE layer is successfully contacted on all members, the FUSE ID is returned to the *CreateGroup* caller.

FUSE state is never orphaned by failures, even when those failures occur just after group creation. An application may receive a FUSE ID from the group creator, and then attempt to associate a failure handler with this FUSE group, only to find out that the group no longer exists because a failure has already been signalled. This causes the failure handler to be invoked, just as if the notification had arrived after the failure handler was registered.

3.3 Liveness Monitoring and Failure Notification

Once setup is complete, our FUSE implementation monitors the liveness of the group members using a spanning tree whose individual branches follow the overlay routes between the group creator and the group members. Each link in the tree is monitored from both sides; if either side decides a link has failed, it ceases to acknowledge pings for the given FUSE group along all its links. When this occurs, one could immediately signal group failure, but our implementation instead attempts repair, as will be explained in more detail in Section 6.

This mechanism allows FUSE to guarantee that any member of the group can cause a failure notification to be received by *every* other live group member. FUSE invokes the failure notification handler exactly once on a node before tearing down the state for that FUSE group. A node hearing the notification does not know whether it was due to crash, network disconnect or partition, or if the notification was explicitly triggered by some group member. Explicit triggering is a necessary component of FUSE because FUSE does not guarantee that it will notice all persistent communication problems between group members automatically; it only guarantees that a communication failure noticed by any group member will soon be detected by all group members.

FUSE only guarantees delivery of failure notifications, and only to nodes that have already been contacted during group creation. Note that FUSE clients cannot use this mechanism to implement general-purpose reliable communication. Therefore, well-known impossibility results for consensus do not apply to FUSE [22, 24]. A concrete example illustrating this limitation is a network partition. FUSE members on both sides of the partition will receive failure notifications, but it is not possible to communicate additional information, such as the cause of the failure, across the partition.

FUSE will sometimes generate a notification to the entire group even though all nodes are alive and the next attempted communication would succeed. We refer to such a notification as a false positive. It is easy to see how false positives can occur – transient communication failures can trigger group notification. The possibility of false

positives is inherent in building distributed systems on top of unreliable infrastructure. One can tune the timeout and retry policy used by the liveness checking mechanism, but there is a fundamental tradeoff between the latency of failure detection and the probability that timeouts generate false positives. We do not provide an API mechanism for applications to modify the FUSE timeout and retry policy. Providing such a mechanism would add complexity to our implementation, while providing little benefit: applications already need to implement their own timeouts, as dictated by their choice of transport layer. FUSE requires this participation from applications because FUSE does not necessarily monitor every link.

Because we implement liveness checking using overlay routes, the maximum notification latency is proportional to the diameter of the overlay: successive failures at adversarially chosen times could cause each link to fail exactly one failure timeout after the previous link. However, we expect notification after a failure to rarely require more than a single failure timeout interval because our FUSE implementation always attempts to aggressively propagate failure notifications. Also, application developers do not need to know the maximum latency in order to specify their timeouts. As mentioned above, sends should be monitored using whatever timeout is appropriate to the transport layer used by the application. If the sender times out, it can signal the FUSE layer explicitly. If a node is waiting to receive a message, specifying a timeout is difficult because only the sender knows when the transmission is initiated. In this case, if the sender has crashed, developers should rely on the FUSE layer timeout to guarantee the failure handler will be called.

FUSE failure notifications do not necessarily eliminate all the race conditions that an application developer must handle. For example, one group member might signal a failure notification, and then initiate failure recovery by sending a message to another group member. This failure recovery message might arrive at the other group member before it receives the FUSE failure notification. In our experience, version-stamping the data associated with a FUSE group was a simple and effective means of handling these races.

3.4 Fail-on-Send

FUSE does not guarantee that all communication failures between group members will be proactively detected. For example, in wireless networks sometimes link conditions will allow only small messages – such as liveness ping messages – to get through while larger messages cannot. In this case, the application will detect that the communication path is not working, and explicitly signal FUSE. We call this reason for explicitly signalling FUSE *fail-on-send*.

There are two categories of failures that require fail-on-send. The first is a communication path that successfully transmits FUSE liveness checking messages but which does not meet the needs of the application. The second

is a failed communication path the application is using, but which FUSE is not monitoring.

An example from this second category is an intransitive connectivity failure. If two applications cannot communicate directly, but are both responding to FUSE messages from a third party, they may only experience a failure upon attempting to exchange a message. FUSE still guarantees that if either party triggers a notification at this point, all live group members will hear a notification.

Some applications may generate mixed acknowledged and un-acknowledged traffic. For example, an application might send streaming video over UDP alongside a control stream over TCP. In this case, it is up to the application to decide which delivery failures warrant a notification. Fail-on-send allows this failure case to be handled in the same manner as the previous cases.

3.5 Failure Model and Security

FUSE is designed to handle node crashes and arbitrary network failures, but not malicious behavior. The application we built using FUSE handles malicious behavior through redundancy above the FUSE layer by using multiple content distribution trees (see Section 4).

FUSE assumes a network failure model consisting of any pattern of packet loss, duplication or re-ordering. This includes simultaneous network partitions and even an adversary dropping packets based on their content. For any network failure, FUSE guarantees that all parties agree whether or not a failure has occurred. Our FUSE implementation routes all FUSE and overlay messages over TCP connections. Our implementation handles arbitrary packet loss and re-ordering, but only handles duplication to the extent that TCP does. It would be straightforward to extend our implementation to handle arbitrary duplication by incorporating digital signatures and timestamps, though we have not yet done so. This extension would also prevent tampering with message contents. FUSE's ability to handle packet loss is not dependent on using a reliable transport layer, such as TCP. Alternative FUSE implementations could use unreliable transport layers, such as UDP. Using a different transport would present different performance characteristics that many application developers would want to be aware of.

Our model for node failures is fail-stop. Software failures that are recognized by the application (e.g., misconfiguration is detected) can be handled by explicitly signalling the FUSE group. FUSE also handles software failures that result in a process exit, such as unhandled exceptions. FUSE does not handle nodes that behave maliciously, either due to explicit compromise or due to software faults that are not appropriately contained.

Malicious nodes can attack FUSE in one of two ways: by dropping legitimate failure notifications or by unnecessarily generating failure notifications. Dropping a failure notification, and then continuing to generate ping messages for the failed group, can delay the notification indefinitely for certain group members. This violates the FUSE

notification semantics. Generating unnecessary failure notifications can prevent the use of otherwise functional FUSE groups, thus leading to a Denial-of-Service (DoS) attack. Of course, if the application response to failure notification is to re-attempt the failed operation with a different set of nodes, sustaining a DoS attack may be quite difficult.

3.6 Crash Recovery

Our implementation of FUSE does not use stable storage, and so crash recovery is trivial. The implementation performs the same actions during crash recovery as during any other initialization.

A recovering node does not know whether a failure notification was propagated to other group members. FUSE handles this case and several other corner cases by having nodes actively compare their lists of live FUSE groups as part of liveness checking. We will discuss the details of our implementation more in Section 6, but the effect is that disagreements about the current set of live FUSE groups are detected within one failure timeout interval. Disagreements are resolved by triggering a notification on any groups already considered to have failed by some group member.

An alternative FUSE implementation could use stable storage to attempt to mask brief node crashes. A node recovering from a crash could assume that all the FUSE groups in which it participates are still alive; the active comparison of FUSE IDs would suffice to reliably reconcile this node with the rest of the world. Furthermore, there is no compatibility issue: Nodes employing stable storage could co-exist with nodes not employing stable storage without any change to the FUSE semantics. It is still the case that a persistent communication failure on the node recovering from crash would cause all the FUSE groups it participates in to be notified. Applications can also make use of volatile-state FUSE groups to guard state stored in stable storage, but this requires additional application-level complexity.

4 Applications

As part of the Herald [7] project to build a scalable event notification service, we have been exploring the construction of scalable, reliable application-level multicast groups using a scalable peer-to-peer overlay network. Grappling with the complexities of implementing failure handling and automatic re-configuration led us to invent a new abstraction for failure notification.

The deciding factor for inventing FUSE was our design of multicast groups. A well-known technique for implementing application-level multicast on an overlay is to construct the multicast trees using reverse path forwarding (e.g., Scribe [37]). One major drawback of this approach is that nodes on an overlay routing path between a subscriber and the root node must forward a potentially large amount of traffic for the multicast group, even if they

have no interest in it. To remove this potential obstacle to deployment, we designed Subscriber/Volunteer (SV) trees [20]. SV trees route content around non-interested parties by establishing separate content-forwarding links among subscribers and volunteers. This leads to two inter-related data structures: a content forwarding tree overlaid on a reverse path forwarding (RPF) tree.

The content-forwarding tree is straightforward to construct in the absence of failure. However, repairing the tree without introducing distributed routing cycles proved difficult in the face of arbitrary and possibly simultaneous node failures, link failures, message loss, and routing changes in the overlay. To manage this complexity, we adopted a simple design pattern: garbage collect out-of-date state using FUSE and retry by establishing a new FUSE group and installing new application-level state. FUSE allowed us to tie together all the distributed state that needed to be garbage collected.

This design pattern drastically reduced the state space that we had to consider and was instrumental in achieving a working SV tree implementation. For example, a single FUSE group ties together the endpoints of a content-forwarding link and all the RPF tree nodes bypassed by that link. Failure notification on this group garbage collects all the relevant state. After failure notification, the subscriber that requested creation of the now-failed content-forwarding link is responsible for creating a replacement FUSE group and forwarding link. If this subscriber is dead, then no replacement is needed. Indeed, there was a natural choice for the FUSE group creator everywhere we used FUSE, obviating the need for a voting mechanism to manage group creation or re-creation.

As mentioned in Section 3.3, FUSE did not eliminate all race conditions in SV trees, but the remaining ones were trivial to handle. For example, subscribers add version stamps to each subscription request to prevent late-arriving FUSE notifications from acting on new content-forwarding links.

FUSE also reduced the amount of code required to implement SV trees. Without FUSE, we would have had to include a large amount of additional context in each message to allow the recipient to garbage-collect now-invalid state and we found it difficult to reason about the correctness of this non-FUSE alternative design. We also used FUSE in one important non-failure case: when a multicast group participant voluntarily leaves the tree, we explicitly signal the FUSE group that would have been signaled if the node had failed. This causes the appropriate repairs to occur, removing the node from the content-forwarding tree.

Our desire to support large multicast trees does not require that we support individual FUSE groups with a large number of members. We designed SV trees to use a large number of small to medium size FUSE groups, and this determined the scalability requirements for FUSE. For example, simulating a 2000 subscriber tree on a 16,000 node overlay required an average of 2.9 members per FUSE

group with a maximum size of 13. We also verified that the maximum and mean FUSE group sizes depend very little on the size of the multicast tree, and increase slowly with the size of the overlay.

4.1 Other Applications

From our experience implementing an event delivery service with FUSE, we believe that many applications built on top of scalable overlay networks can benefit from the use of FUSE. Many of these applications construct a large number of trees, and then monitor parent-child links in these trees using application-level heartbeats. Using FUSE for these application-level heartbeats would allow liveness checking traffic to be shared across all the trees.

Another type of application where FUSE would be useful is a Content Delivery Network (CDN) that replicates a large number of documents and pushes updates to them. If the replication topology varies on a per-document basis, this will entail a large number of replication multicast trees. A common strategy for reliability on these trees mirrors the approach discussed above for peer-to-peer applications: heartbeats ensure that each replica site for a given object can track whether it is receiving all updates correctly, or is instead somehow disconnected from the tree. FUSE can replace the per-tree heartbeat messages with a more efficient and scalable means of detecting when the trees need to be reconfigured due to node or network failures.

Peer-to-peer storage systems such as TotalRecall [4] and Om [45] could also benefit from the efficiencies of using FUSE to implement liveness checking. For example, TotalRecall relies on the overlay for liveness-checking of eager replicas, but must separately implement liveness-checking of lazy replicas. The substitution of FUSE groups would be straightforward. Om implements its own failure detection and timeout scheme using leases; these leases could be replaced by FUSE groups. FUSE would also be a good fit for Om because every replica in Om can regenerate the entire replica set, and therefore should monitor the liveness of all other replicas. This symmetric responsibility exactly corresponds to the semantics of FUSE group notifications. Lastly, the potential for false positives using FUSE does not compromise Om's consistency guarantees; Om's failure-induced reconfiguration protocol is already designed to be robust to failure-detection false positives.

In addition, FUSE may also be useful in some of the application areas targeted by weakly consistent membership services. For example, Vogels and Re [44] argue that weakly consistent membership services would benefit many Web Services, ranging from scientific computing to federated business activities. FUSE may be a more suitable choice for some of these emerging applications.

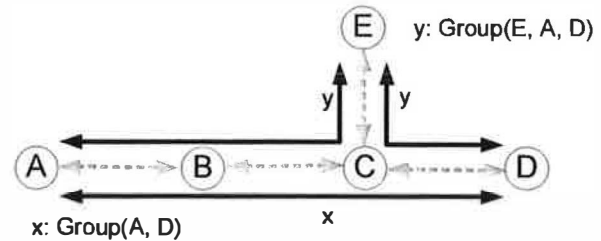


Figure 2. Two FUSE groups being monitored by overlay pings. The black lines denote end-to-end checking, while dashed gray lines denote active overlay pings.

5 Liveness Checking Topologies

Many different liveness checking topologies can be used to implement FUSE. In this Section, we describe in detail the topology we chose: per-group spanning trees on an overlay network. We then discuss other topologies and their security/scalability tradeoffs.

Our overlay design requires a content-addressable overlay (e.g., DHTs such as CAN, Chord, Pastry, Skip-Net, or Tapestry [25, 34, 36, 39, 46]). Figure 2 depicts an overlay topology with two live FUSE groups, x and y . The spanning tree for FUSE group y contains the group members A , D , and E , and two additional nodes, B and C , that we refer to as *delegates*. Delegates arise because the FUSE liveness checks are routed along overlay paths between members, and these paths may contain nodes that are not members. If overlay routes change or delegates fail, delegates may be added to or deleted from the liveness checking tree; this repair process is explained in detail in Section 6.

Building liveness checking trees on top of an overlay lets us reuse the liveness checks that the overlay uses to maintain its routing tables. The liveness checking tree for a given FUSE group is the union of the overlay routes between the group creator (the *root*) and the group members. When multiple FUSE groups have overlapping trees, each overlay ping message monitors all FUSE groups whose liveness checking trees include that overlay link. Figure 2 illustrates this sharing for the FUSE groups x and y .

In the absence of failures, our FUSE implementation requires no additional messages beyond the overlay pings to monitor FUSE groups. Only group setup, teardown, and repair incur per-group costs. This allows one to build systems that require very large numbers of FUSE groups.

5.1 Alternative Topologies

In this section, we present three alternative topologies for FUSE liveness monitoring that provide better security guarantees at the cost of worse scalability. Certain implementations of these topologies can be simpler and provide stronger guarantees for worst-case failure notification latency.

As mentioned in Section 3.5, malicious nodes can mount two kinds of attacks against FUSE: the dropped

notification attack and the unnecessary notification attack. In the overlay topology used by our FUSE implementation, these attacks can be mounted by malicious group members or delegates. The SV tree application handles the dropped notification attack above the FUSE layer by using redundant content-distribution trees. It handles the unnecessary notification attack by re-creating FUSE groups with different sets of members.

The first alternative topology we consider is per-group spanning trees *without* an overlay. By routing liveness checking traffic directly between members, this topology eliminates the threat of delegates launching attacks on FUSE. The scalability tradeoff for this additional security is that the overhead of liveness checking traffic may be additive in the number of FUSE groups – the opportunities to share liveness checking traffic will depend on the degree of overlap in FUSE group membership.

The second alternative topology we consider is per-group all-to-all ping (again, without an overlay). This improves security even further; all-to-all ping is robust to dropped notification attacks from members because no member relies on any other node to forward failure notifications. However, this topology requires n^2 messages for a group of size n – significantly more than the per-group spanning tree topology. An added benefit of the all-to-all topology is that worst-case failure notification latency is reduced to twice the ping interval.

The final topology we consider is using a central server to ping all nodes. This may be an appropriate topology for using FUSE within a data center environment. From a security standpoint, this server represents a single point of trust, which may be easier to secure than a larger collection of machines. If the server is compromised, attacks can be launched against any FUSE group in the system. If not, the security guarantees are the same as in the all-to-all ping topology. For settings that span multiple administrative domains, the use of a single trusted server may not be appropriate. The scalability of this topology is limited: all FUSE traffic passes through the server, which can be a bottleneck for a large number of FUSE participants. However, the load on each group member is minimal: each group member only pings the central server during each ping interval.

6 Implementation

The key architectural choice we faced in implementing FUSE was whether to route all FUSE messages using the overlay paths, or to route certain messages directly between the group members. In the topology we used, spanning trees along overlay routes, path failures involving delegates can be dealt with in one of two ways. One option is to signal a failure on all FUSE groups using that path. This has the advantage of implementation simplicity, but can be a significant source of false positives. Instead, we chose the second option: to attempt to repair the liveness monitoring topology for the group.

Repair will succeed if the members of the group can still communicate with each other directly, and therefore repair routes around all failures involving delegates. We chose to implement repair by routing repair messages directly from the root to the group members. The overriding factor for this choice was rapidity of failure detection: relying solely on overlay routes would require waiting for the overlay to attempt to repair itself before signaling a failure. Using direct root to member communication allows failures involving group members to be detected more rapidly. This direct communication also results in better latencies for group creation and application-signalled failure notifications. When overlay routing paths are working, we still get the scalability benefits of shared spanning trees using the overlay. In the remainder of this section, we first describe the functionality exposed by the SkipNet overlay, and we then discuss the details of implementing each FUSE operation.

6.1 Overlay Functionality

Our implementation of FUSE on top of the SkipNet overlay required two features that SkipNet provides to client applications: messages routed through the overlay result in a client upcall on every intermediate overlay hop, and the overlay routing table is visible to the client. This functionality is standard for many overlays [18].

Our FUSE implementation re-uses the overlay routing table maintenance traffic by piggybacking a SHA1 hash (20 bytes) on ping requests. This hash encodes all the FUSE groups that use this overlay link. FUSE could have sent its own messages across these same links, but the piggybacking approach amortizes the messaging costs. SkipNet pings cause a client upcall at their destination, so the destination FUSE layer can examine the piggybacked contents. Because all SkipNet links are monitored from both sides, we did not need to add additional ping at the FUSE layer to ensure this. Implementing FUSE on an overlay that does not have these properties would require FUSE to perform additional ping itself.

All FUSE and overlay messages in our system are delivered over TCP, and therefore inherit TCP's retry and congestion control behaviors. When a TCP connection breaks, or a liveness checking message fails to get through before the timeout, we interpret that to mean that the node at the other end is unavailable.

6.2 Group Creation

We implement group creation as follows: Group creation does not finish until every member node has a timer installed that will signal failure in the event of future communication failures. These timers are only reset by the receipt of liveness checking messages. Thus, any future communication failures will be converted into failure notifications.

To achieve low creation latencies, the creating node directly contacts every other member node in parallel,

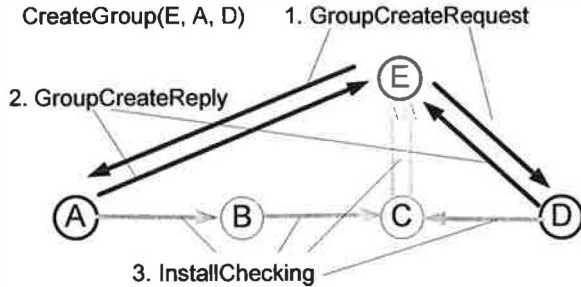


Figure 3. *Group Creation Example.* Root node E sends *GroupCreateRequest* messages directly to group members A and D. Nodes A and D reply directly with *GroupCreateReply* messages, and also route *InstallChecking* messages through the overlay towards E.

declaring creation to have succeeded when all of them have replied. When *CreateGroup* is called on a node, that node (referred to as the root) generates a unique ID for the group. The root also creates an entry in its list of groups being created, and associates a timeout with this group creation attempt. The entry contains the FUSE ID, the list of group members, and which members the root has received a reply from. The root then sends *GroupCreateRequest* messages to the other member nodes. An example message sequence that could result from group creation is shown in Figure 3.

On receiving a *GroupCreateRequest*, a member node installs FUSE member state for the group: the unique ID, a sequence number that is initially 0 (and which is incremented by group repair), and the identity of the root. Concurrently to sending the *GroupCreateReply* directly to the root, the member node routes an *InstallChecking* message towards the root using overlay routing. The *InstallChecking* message will set a timer on every node it encounters to ensure that liveness checks are heard.

If the root receives a *GroupCreateReply* from every member within the group creation attempt timeout, it installs the FUSE root state for the group: the unique ID, the sequence number, the identities of all the other group members, and a timer for checking that *InstallChecking* messages have arrived from every member. The root then removes this group from its list of groups being created and returns the unique ID to the FUSE client application.

If the *GroupCreateReply* is not received from every node within the group creation attempt timeout, the group creation fails and the root returns failure to the FUSE client application. The root also attempts to send a failure notification for this FUSE group to each group member. This notification is a *HardNotification* – we elaborate on the different types of notifications in Section 6.4. Finally, the root removes this group from its list of groups being created. This prevents *GroupCreateReply* messages received later from causing installation of state for the failed group creation.

When an *InstallChecking* message arrives at a delegate

node, it installs the FUSE delegate state for the group: the FUSE ID, sequence number, and current time are associated with both the previous hop and the next hop of the *InstallChecking* message, and timers are associated with both hops as well. The node then forwards the message towards the root. If the timer for receiving all the *InstallChecking* messages fires on the root, the root attempts a repair.

6.3 Steady-State Operation

Whenever an overlay node initiates a ping to a routing table neighbor, it piggybacks a hash of the list of FUSE IDs that this node believes it is jointly monitoring with its neighbor. When the neighbor receives this message, if the hash matches, the neighbor resets the timers for all the (FUSE ID, neighbor) pairs represented by the hash. There can be more than one timer per FUSE ID because a node may have more than one neighbor in the liveness checking tree. If one of these timers ever fires, the node sends a *SoftNotification* message to every neighbor in the liveness checking tree for this FUSE group, and then it cleans up the FUSE delegate state for the group. Additionally, if the timer is firing on a member, a repair is initiated.

If a node receives a non-matching hash of FUSE IDs from a neighbor, both nodes attempt to reconcile the difference by exchanging their lists of live FUSE IDs. If they can communicate, they only remove the liveness checking trees on which they disagree, and the timers are reset on the others. If they cannot communicate, the relevant checking state is removed, and *SoftNotification* messages are sent.

During group creation, a race condition exists that can cause hash mismatches: a node that has just received an *InstallChecking* message may receive a ping from the next hop of the *InstallChecking* message. We resolve this race condition using a brief grace period. A node only removes a liveness checking tree that its neighbor does not believe exists if that tree has existed for longer than the grace period; in our implementation, this period is 5 seconds.

6.4 Notifications

To achieve the simultaneous goals of low notification latency and resilience to delegate failures, our FUSE implementation distinguishes between different classes of failures. Failures of the steady-state liveness checking trigger a *SoftNotification*. This message is distributed throughout the liveness checking tree, which alerts the root that a repair is needed and prevents a storm of *SoftNotifications* from being sent to the root by the rest of the tree. Members receiving a *SoftNotification* also initiate repair directly with the root as described in Section 6.5.

Failures of group creation or group repair trigger a *HardNotification*. Because both create and repair use direct root-to-member communication, delegate failures do not incur false positives. Note that *SoftNotifications* do not cause failure notifications at the application layer. In-

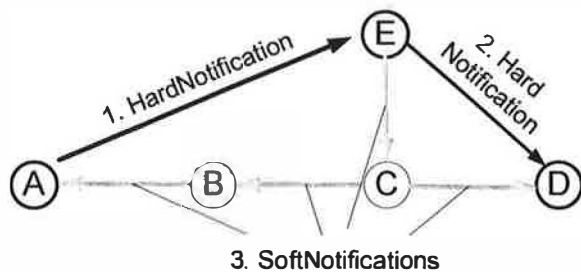


Figure 4. Explicitly Signalled Notification Example. *SignalFailure()* is called on node A. Node A sends a *HardNotification* to the root, E. E forwards the *HardNotification* to the remaining group member, D. E also generates a *SoftNotification* to clean up the liveness checking tree.

stead, they trigger repair actions. The failure of these repair actions will lead to a *HardNotification*, which is reflected at the application layer. To achieve low latency for explicitly signalled notifications, *HardNotifications* are also used to convey them.

A member generating a *HardNotification* sends it to the root, which in turn forwards it to all other group members. A node receiving a *HardNotification* immediately invokes the application-installed failure handler. The root node additionally sends *SoftNotifications* to proactively clean up the liveness checking tree. An example of such a message sequence is shown in Figure 4.

A node receiving a *SoftNotification* message first checks to see that the sequence number is greater than or equal to its recorded sequence number for the specified group; recall that the sequence number is incremented during the repair process. If not, the message is discarded. If the sequence number is current, the node forwards the message on to all neighbors in the liveness checking tree other than the message originator, and removes its delegate state for the group. If the node is a member or the root, it also initiates repair.

6.5 Group Repair

When a member initiates a repair, it sends the root a *NeedRepair* message, and installs a timer for hearing back from the root. If the timer fires, it signals a failure notification to the FUSE client application, sends a *HardNotification* message to the root, and cleans up the state associated with this group.

The root can be signalled that a repair is needed through either of two paths: a *NeedRepair* directly from a member or a *SoftNotification* spreading through the liveness checking tree. The *NeedRepair* message is needed to remove a potential source of variability in the latency of repair. When a member receives a *SoftNotification*, it does not have a good estimate for how soon the root will similarly receive a *SoftNotification*; these notifications are routed through the overlay, and breaks in overlay routing require a timeout on the other side to continue the progress of the *SoftNotification*. An example of a se-

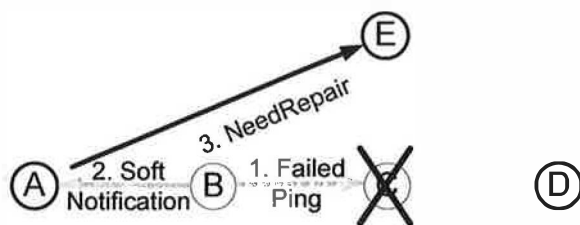


Figure 5. Messages Triggering Group Repair Example. Delegate B sends a ping to delegate C, and the ping is not acknowledged. B then sends a *SoftNotification* to member A. A then sends a *NeedRepair* to root E. Once E has been alerted, E coordinates repair just as it coordinated creation.

quence of messages leading to a *NeedRepair* is shown in Figure 5.

When the root attempts a repair, it sends out *GroupRepairRequest* messages to every member. Members answer these with *GroupRepairReply* messages, and they send *InstallChecking* messages just as in group creation. State management at the root during repair is similar to creation, involving a repair attempt table where open repairs are recorded. State management at member nodes is different: if a repair message ever encounters a member that no longer has knowledge of the group, it fails and signals a *HardNotification*. This guarantees that repairs will not suppress any *HardNotification* that has already reached some members. Such notifications garbage collect all group state at the node.

Nodes receiving a *GroupRepairRequest* increment the group sequence number so that late-arriving *SoftNotification* messages will not trigger a redundant repair. If the root decides that repair has failed (using the same criterion as for create failing), the root sends *HardNotifications* to all members, and signals the application.

As we discussed in Section 5, using overlay paths allows us to achieve low steady-state message overheads. We believe a repair scheme that better localizes repair traffic is possible, but we did not consider this a case worth optimizing. During transient overlay routing failures, repairs may be quite frequent; a node consulting its routing table may learn that there is no next hop for an *InstallChecking* message. To reduce the message volume under such circumstances, we implement per-group exponential backoffs (capped at 40 seconds) for the frequency of repairs. Although repair generates additional network traffic shortly after a failure has been detected, our use of TCP serves to regulate this additional network load.

7 Experimental Evaluation

We evaluate FUSE running on top of the SkipNet [25] overlay network using two main techniques: a scalable discrete event simulator and a live implementation with up to 400 virtual nodes running on a cluster of 40 workstations. Our SkipNet and FUSE implementations running

on the live system and in the simulator use an identical code base, except for the base messaging layer.

7.1 Methodology

We configured the SkipNet overlay to employ a 60 second ping period, a base of size 8, and a leaf set of size 16. For a 400 node overlay, this yielded an average of 32.3 distinct neighbors per node.

For the cluster evaluation our router uses ModelNet [41] to emulate wide-area Internet-like network characteristics. We ran 10 processes on each of the 40 physical nodes, for a total of 400 virtual nodes. In order to emulate nodes running on physically separate machines, there is no explicit state sharing between these processes, and all communication between processes is forced to pass through ModelNet.

The motivating scenario for our choice of router topology and our assignment of link latencies and bandwidths was small and large corporations on multiple continents with direct Internet connections. Both our live and simulator experiments were run on a Mercator topology [40] with 102,639 nodes and 142,303 links. Each node is assigned to one of 2,662 Autonomous Systems (ASs). There are 4,851 links between ASs in the topology. We assigned 97% of links to be OC3 and 3% to be T3. For each OC3 link, we assigned the link latency uniformly between 10 and 40 milliseconds, and we assigned a bandwidth of 155 Mbps. For each T3 link, we assigned the link latency uniformly between 300 and 500 milliseconds, and we assigned a bandwidth of 45 Mbps. This led to round-trip latencies with a median value of 130 milliseconds, and a significant heavy-tail. In Figure 6, the curve labeled Simulator shows a CDF of end-to-end latencies; paths crossing one or more T3 links are in the heavy-tail.

We also used this topology in our simulator, where we ran experiments with both 400 and 16,000 nodes, to model how our system would scale to a much larger deployment. The simulator used the same latency values, but did not model bandwidth constraints.

Our event notification service workload (Section 4) creates a large number of groups with an average group size of less than 3. Even scaling up to a 16,000 node overlay in our simulator, the maximum group size we observed was 13. Just as these results informed our FUSE design, they also determined our choice of evaluation parameters: we evaluated FUSE on a workload of groups ranging from 2 to 32 members.

7.2 Calibration of Simulator and ModelNet

We used an experiment that performed RPC message exchanges between randomly chosen nodes on a 400-node overlay network to calibrate the wide-area network topology model used in our experiments and to make sure that results obtained through simulation were comparable to those obtained through running on the live cluster with ModelNet.

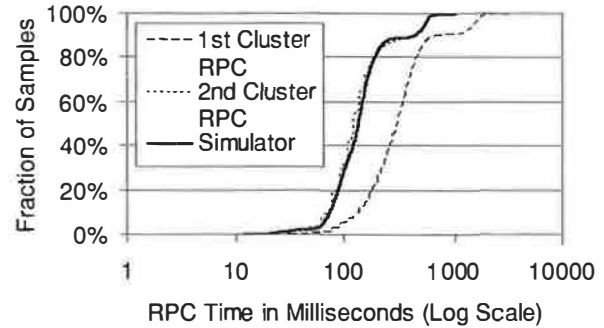


Figure 6. RPC Latencies

Towards that end, we measured 2400 RPCs on both our cluster and our simulator. Figure 6 shows a Cumulative Distribution Function (CDF) of the RPC times measured for three sets of RPCs: those obtained in the simulator, and two kinds of RPC times obtained on the cluster. Because the cluster code caches TCP connections between pairs of nodes, the first communication between a pair of nodes takes longer than subsequent communications, due to the additional time required for connection establishment. Our experiment performs two back-to-back RPCs between pairs of nodes on the cluster and reports the durations of both the first RPC, which is likely to incur connection setup overhead, and the second one, which will not.

As can be seen in Figure 6, the values for the second RPC on the cluster closely track those for the simulator. This gives us confidence that both the simulator and ModelNet are faithfully modeling the chosen Mercator topology.

7.3 FUSE Group Creation Latencies

We measured the time on the cluster required to create a FUSE group as a function of group size when group members are uniformly distributed throughout the system. We used group sizes 2, 4, 8, 16, and 32 and created 20 groups of each size. Figure 7 shows the results. While group members were contacted in parallel, the more members there are, the greater the chance of including nodes at a significant network distance from the root node of the group. Note, for instance, that for groups of size 8, the 75th percentile time was significantly larger than the median, whereas for groups of size 32, the 25th percentile, median, and 75th percentile are all relatively close, as with this many members the chance of encountering one or more slow communication paths is quite high.

In the simulator, we evaluated both a 400 node and a 16,000 node system. The simulated creation times followed the same pattern as those on the cluster, except that they tended to be about half as long, for the same reasons that the simulated and actual RPC times in Figure 6 for new connections also differed by about a factor of two. The group creation times for the simulated 16,000 node system were essentially identical to those for the 400 node

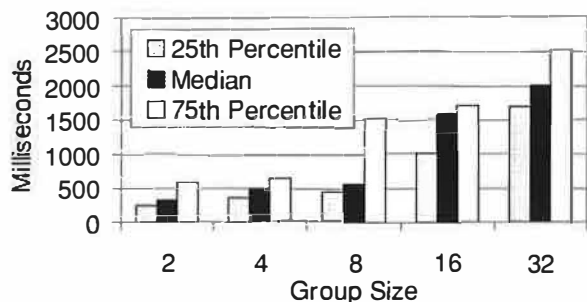


Figure 7. Latency of Group Creation

simulated system. This is to be expected, since creation messages are routed directly between the root and members, and therefore are not affected by the length of overlay routes.

7.4 Failure Notification Latencies

The latency of failure notification for an actual failure is comprised of two parts: the time for a node in the system to decide that a failure has occurred, and the time for FUSE to propagate that information to all group members. The time to detect a failure depends on the type of failure, and on how node and link failures are monitored. We perform two experiments to characterize these costs: our first experiment investigates the latency of explicitly signaled failures, and our second investigates the latency of failure notification when nodes crash.

To measure the notification latency of explicitly signaled failures, we chose a group member at random from the same set of groups used for the group creation experiments, and had it explicitly signal failure. Figure 8 shows the notification times over 20 such create/notify cycles. As expected, the notification latencies are significantly lower than the group creation latencies. This improvement is a result of three factors. First, our messaging layer maintains a cache of recently used TCP connections rather than opening a new TCP connection each time a message is sent. During this experiment, the failure notifications travel over cached TCP connections because these connections were recently used to perform group creation. Second, failure notifications only require a one-way message, not a round-trip. Third, creation blocks at the root until all members have been contacted, and thus a single node in the group that is far away in the network will delay the entire create operation. In contrast, notification takes effect at each member as soon as the notification has arrived. The maximum notification time observed for any FUSE group was 1165 ms. Our simulation results at 400 nodes matched the results obtained on the cluster. We also investigated scaling behavior in the simulator, and we found the expected result that notification times did not increase in a 16,000 node overlay.

Even though failure notifications take effect at each member upon arrival, in Figure 8 we see that the median notification latency shows a dependence on the group size. The rise in the curve at group sizes 2, 4, and 8 is due

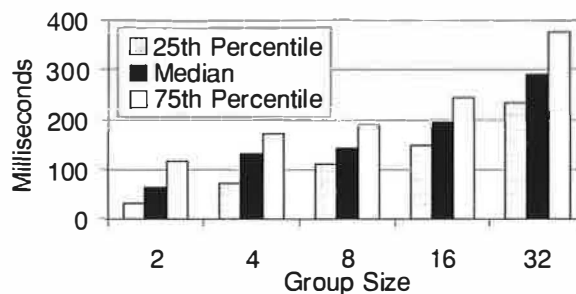


Figure 8. Latency of Signaled Notification

to the extra forwarding hop needed when notifications are generated by a non-root node. At size 2, these notifications just travel from the member to the root, whereas at sizes 4 and 8 notifications travel from the member to the root and then to all other members. The additional increase in notification latency for groups of size 16 and 32 reflects the latency added by our messaging layer at the root. Our implementation uses an XML-based messaging system with high message serialization overhead: reducing this overhead would be straightforward but this was not the focus of our work. We ran micro-benchmarks and determined that running 10 virtual nodes on each physical machine adds approximately 1.1 ms of overhead per message, and the base overhead of a message send including XML serialization is 2.8 ms.

To measure the latency of failure notification when nodes crash, we performed the following experiment: we created 400 FUSE groups of size 5 and then disconnected the network on one of the 40 physical machines, disconnecting 10 of the 400 virtual nodes. Of the 400 FUSE groups, 42 contained one or more disconnected virtual nodes as members. All remaining members of these groups received failure notifications – a total of 163 notifications.

Figure 9 shows the distribution of these notification times. These times have several components: the time until a ping of the failed node is attempted, the timeout for this ping, the time for a member to learn of the failed ping, the time for the subsequent repair attempt to fail, and the actual notification time after repair has failed. We used a ping interval of one minute, and a ping timeout of 20 seconds, so the total ping timeout latency should be uniformly distributed between 20 and 80 seconds. If a member has failed, the root times out after 2 minutes with no repair response. If a root has failed, the members time out after 1 minute with no repair response, leading to shorter overall failure notification times. Figure 8 shows that the notification times are less than a second. We deduce from this that the ping and repair timeouts dominate the failure notification times in the event of a node crash.

7.5 Steady State Load and Churn

One set of experiments we performed measured the amount of background network traffic present due to the overlay network and due to FUSE groups that are using

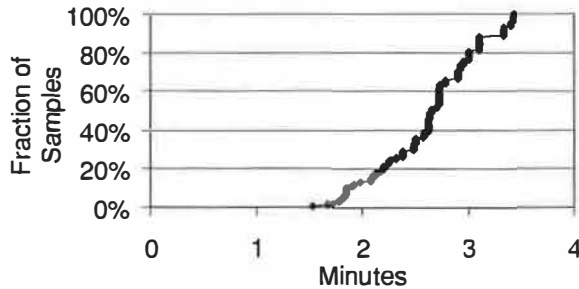


Figure 9. Combined Latency of Ping Timeout, Repair Timeout, and Failure Notification. Ping and Repair Timeouts dominate other factors.

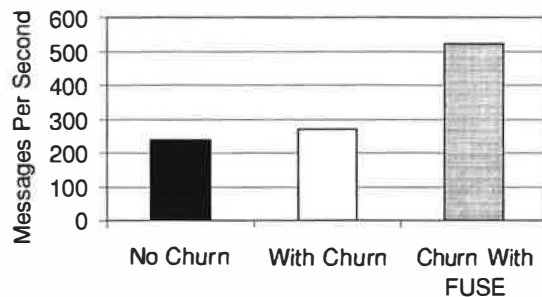


Figure 10. Costs of Overlay Churn

the overlay network for liveness checking. On the cluster, we observed background network traffic loads of 337 messages per second over a 10 minute interval when no FUSE groups were present and 338 messages per second over a subsequent 10 minute interval when 400 FUSE groups of 10 members each were present. These experiments verified that, in the absence of node failures, FUSE groups imposed no additional messages beyond that already imposed by the overlay itself; the only additional cost was a 20 byte hash piggybacked on each ping.

When nodes are entering and leaving the overlay network (often referred to as “churn”), the overlay paths used for liveness checking between nodes in a FUSE group may change, causing the liveness checking state to have to be reconstructed. Overlay churn does not cause false positives in FUSE, but it does cause FUSE to generate higher network load. We experimentally quantified the network load imposed by a high churn rate with FUSE groups.

We designed our churn experiment to use a very aggressive rate of arrivals and departures. We used 200 stable nodes that remained alive for the duration of the experiment and 200 nodes that were killed and restarted such that an average of 100 of the churning nodes were alive at any given time. The rate of churn resulted in a system-wide average half-life of 30 minutes. This is more than a factor of 7 higher rate of churn than was observed in a 2003 study of the OverNet peer-to-peer system [3]. We created a total of 100 FUSE groups of size 10 on the 200 stable nodes, so that on average each stable node was a member of five FUSE groups.

We measured both CPU loads and network message traffic. CPU loads did not show noticeable increases dur-

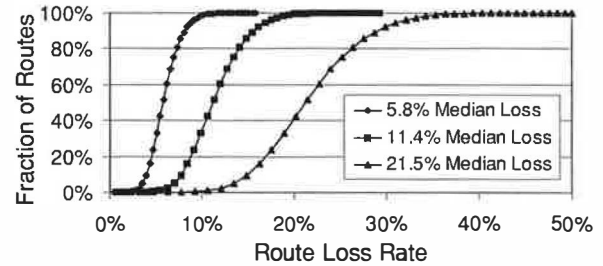


Figure 11. CDFs of Per-Route Loss Rates for Three Different Per-Link Loss Rates

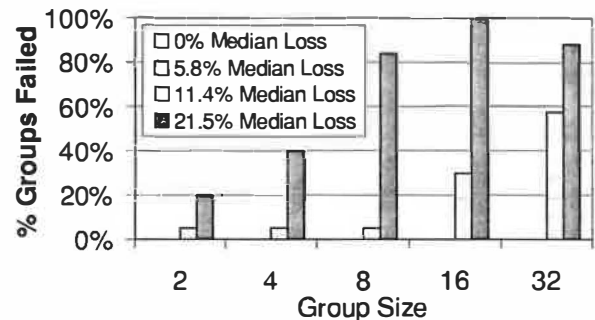


Figure 12. Group Failures Due to Packet Loss. No failures occurred for 0% and 5.8% loss rates.

ing overlay churn. As a basis of comparison, a stable 300-node overlay with no FUSE groups generates a load of 238 messages per second. A 400-node overlay network with churn as described above (which results in an average of 300 live nodes at any given time) generates a load of 270 messages per second – a 13% increase due to the costs of repairing the overlay. Adding the 100 10-member FUSE groups to the churning overlay results in a total of 523 messages per second – a 94% increase over the cost of the churning overlay without FUSE groups. These results are displayed in Figure 10. This additional load is caused by the group repair traffic described in Section 6, and is proportional to the number of groups times the average group size. While the overlay routes are in flux, new liveness checking paths cannot be installed and thus the repair mechanism will be triggered repeatedly. One could reduce the FUSE overhead during churn by employing a more proactive repair strategy at the overlay level.

A high enough rate of churn may cause overlay routing to fail entirely. In this case, the FUSE liveness checking traffic will still be proportional to the number of groups times the average group size. After reaching steady state, each root node will periodically ping each group member directly with a *GroupRepairRequest* message.

7.6 False Positives

We studied the robustness of our implementation to false positives stemming from two different sources, delegate failures and unreliable communication links. In the previously described churn and node crash experiments, many groups experienced delegate failures and had to

perform repairs. In both these experiments, notifications were only delivered for groups where a member crashed: delegate failures never led to a false positive.

To understand the impact of potentially unreliable links on FUSE, we ran a set of experiments with ModelNet configured to probabilistically drop packets on a per-link basis. Routes in our topology ranged from 2 to 43 hops with a median of 15. Figure 11 shows the CDFs of per-route loss rates for three different experiments where we varied the per-link loss rates over 0.4%, 0.8%, and 1.6%. The CDFs are labeled by their median end-to-end loss rates. We created 20 FUSE groups each of sizes 2, 4, 8, 16, and 32. We then enabled losses, and allowed the system to run for an additional 30 minutes.

Figure 12 shows the number of FUSE group failures we observed at each loss rate. No false positives occur at the 0% or 5.8% per-route median loss rates because TCP masks drops at the lower loss rates through retransmissions. At higher loss rates some groups did fail; TCP sockets will break under such adverse network conditions. If one desired a FUSE implementation that continued to monitor links under these conditions, an alternative messaging layer should be employed.

8 Conclusions

This paper has presented FUSE – a lightweight distributed failure notification facility. FUSE provides a novel abstraction, the FUSE group, that is targeted at wide-area Internet applications. The FUSE group abstraction provides distributed application developers with a simple programming paradigm for handling failure: failure notifications never fail, and failures are with respect to groups, not individual members. One significant advantage of the FUSE abstraction is that detecting failures is a shared responsibility between FUSE and the application. This allows applications to implement their own definitions of failure, extending the applicability of failure management services.

We implemented FUSE using a peer-to-peer overlay network and evaluated its behavior on a cluster of workstations under a variety of conditions, including node failures, packet loss, and overlay churn. Our evaluation showed that our FUSE implementation is lightweight and can scale to large numbers of moderate size FUSE groups. Our implementation scales by reusing overlay maintenance traffic to also perform liveness checking of FUSE groups, thereby imposing no additional traffic in the absence of node failures. Because the FUSE abstraction can be implemented efficiently, it may find application in domains where consensus-style protocols have proven to be too heavyweight.

FUSE simplifies the complex task of handling failures in distributed applications. We described our experiences building scalable, reliable application-level multicast groups using FUSE, and how FUSE made this task

easier. We believe that the use of FUSE will likewise simplify the construction of other distributed systems.

Acknowledgements

We thank Ken Birman, Jon Howell, Patricia Jones, Keith Marzullo, Stefan Saroiu, Amin Vahdat and Geoff Voelker for their comments on earlier drafts of this paper. We thank Ashwin Bharambe for his insight on integrating ModelNet. We also thank our shepherd, Greg Ganger, and the anonymous reviewers for their insightful feedback.

References

- [1] M. K. Aguilera, W. Chen, and S. Toueg. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In *Workshop on Distributed Algorithms*, pages 126–140, 1997.
- [2] T. Anker, D. Breitgand, D. Dolev, and Z. Levy. Congress: CONnection-oriented Group-address RESolution Service. Technical Report TR 96-23, Hebrew University of Jerusalem, 1996.
- [3] R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *2nd Intl. Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.
- [4] R. Bhagwan, K. Tati, Y.-C. Cheng, S. Savage, and G. M. Voelker. TotalRecall: System Support for Automated Availability Management. In *1st NSDI*, 2004.
- [5] K. Birman, R. van Renesse, and W. Vogels. Adding high availability and autonomic behavior to web services. In *Intl. Conference on Software Engineering (ICSE)*, 2004.
- [6] K. P. Birman. *Reliable Distributed Systems: Technologies, Web Services, and Applications*. Springer-Verlag, Scheduled for 2004.
- [7] L. F. Cabrera, M. B. Jones, and M. Theimer. Herald: Achieving a Global Event Notification Service. In *HotOS*, 2001.
- [8] R. Callon. RFC 1195. Use of OSI IS-IS for Routing in TCP/IP and Dual Environments. Dec. 1990.
- [9] G. Candea, M. Delgado, M. Chen, F. Sun, and A. Fox. Automatic Failure-Path Inference: A Generic Introspection Technique for Software Systems. In *IEEE Workshop on Internet Applications (WIAPP)*, 2003.
- [10] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *3rd OSDI*, 1999.
- [11] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. In *PODC*, 1992.
- [12] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.
- [13] M. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based Failure and Evolution Management. In *1st NSDI*, 2004.
- [14] M. Chen, E. Kiciman, A. Accardi, A. Fox, and E. Brewer. Using Runtime Paths for Macroanalysis. In *HotOS*, 2003.
- [15] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Systems. In *DSN*, 2002.
- [16] M. Chen, A. Zheng, J. Lloyd, M. Jordan, and E. Brewer. A Statistical Learning Approach to Failure Diagnosis. In *Intl. Conference on Autonomic Computing (ICAC)*, 2004.

- [17] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. In *DSN*, 2000.
- [18] F. Dabek, B. Zhao, P. Druschel, and I. Stoica. Towards a common API for structured peer-to-peer overlays. In *2nd Intl. Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.
- [19] A. Das, I. Gupta, and A. Motivala. SWIM: Scalable Weakly consistent Infection-style process group Membership protocol. In *DSN*, 2002.
- [20] J. Dunagan, N. J. A. Harvey, M. B. Jones, M. Theimer, and A. Wolman. Subscriber/Volunteer Trees: Polite, Efficient Overlay Multicast Trees. <http://research.microsoft.com/sn/Herald/papers/SVTree.pdf>, Submitted for publication, 2004.
- [21] P. Felber, X. Defago, R. Guerraoui, and P. Oser. Failure Detectors as First Class Objects. In *Intl. Symposium on Distributed Objects and Applications*, 1999.
- [22] M. Fischer, N. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.
- [23] I. Gupta, T. Chandra, and G. Goldszmidt. On Scalable and Efficient Distributed Failure Detectors. In *PODC*, 2001.
- [24] J. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37:549–587, 1990.
- [25] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *Fourth USENIX Symposium on Internet Technologies and Systems (USITS)*, 2003.
- [26] P. Ji, Z. Ge, J. Kurose, and D. Towsley. A Comparison of Hard-state and Soft-state Signaling Protocols. In *SIGCOMM*, 2003.
- [27] C. Labovitz and A. Ahuja. Experimental Study of Internet Stability and Wide-Area Backbone Failures. In *Fault-Tolerant Computing Symposium (FTCS)*, 1999.
- [28] L. Lamport. Using Time Instead of Timeout for Fault-Tolerant Distributed Systems. *ACM TOPLAS*, 6:254–280, 1984.
- [29] L. Lamport. The Part-Time Parliament. *ACM TOCS*, 16:133–169, 1998.
- [30] B. Liskov. Distributed Programming with Argus. *CACM*, 31:300–312, 1988.
- [31] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. In *SIGCOMM*, 2002.
- [32] J. Mogul, L. Brakmo, D. E. Lowell, D. Subhraveti, and J. Moore. Unveiling the Transport. In *HotNets*, 2003.
- [33] J. T. Moy. *OSPF: Anatomy of An Internet Routing Protocol*. Addison-Wesley, 1998.
- [34] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *SIGCOMM*, 2001.
- [35] T. L. Rodeheffer and M. D. Schroeder. Automatic Reconfiguration in Autonet. In *SOSP*, 1991.
- [36] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.
- [37] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. In *Third Intl. Workshop on Networked Group Communications*, 2001.
- [38] P. Stelling, C. Lee, I. Foster, G. von Laszewski, and C. Kesselman. A Fault Detection Service for Wide Area Distributed Computations. In *High Performance Distributed Computing*, 1998.
- [39] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *SIGCOMM*, 2001.
- [40] H. Tangmunarunkit, R. Govindan, S. Shenker, and D. Estrin. The Impact of Routing Policy on Internet Paths. In *Infocom*, 2001.
- [41] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *5th OSDI*, 2002.
- [42] R. van Renesse, Y. Minsky, and M. Hayden. A Gossip-Based Failure Detection Service. In *Middleware*, 1998.
- [43] W. Vogels. World wide failures. In *ACM SIGOPS European Workshop*, 1996.
- [44] W. Vogels and C. Re. Ws-membership - failure management in a web-services world. In *Intl. World Wide Web Conference (WWW)*, 2003.
- [45] H. Yu and A. Vahdat. Consistent and Automatic Replica Regeneration. In *1st NSDI*, 2004.
- [46] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-Resilient Wide-area Location and Routing. Technical Report UCB//CSD-01-1141, UC Berkeley, 2001.

PlanetSeer: Internet Path Failure Monitoring and Characterization in Wide-Area Services

Ming Zhang, Chi Zhang, Vivek Pai, Larry Peterson, and Randy Wang
Department of Computer Science
Princeton University

Abstract

Detecting network path anomalies generally requires examining large volumes of traffic data to find misbehavior. We observe that wide-area services, such as peer-to-peer systems and content distribution networks, exhibit large traffic volumes, spread over large numbers of geographically-dispersed endpoints. This makes them ideal candidates for observing wide-area network behavior. Specifically, we can combine passive monitoring of wide-area traffic to detect anomalous network behavior, with active probes from multiple nodes to quantify and characterize the scope of these anomalies.

This approach provides several advantages over other techniques: (1) we obtain more complete and finer-grained views of failures since the wide-area nodes already provide geographically diverse vantage points; (2) we incur limited additional measurement cost since most active probing is initiated when passive monitoring detects oddities; and (3) we detect failures at a much higher rate than other researchers have reported since the services provide large volumes of traffic to sample. This paper shows how to exploit this combination of wide-area traffic, passive monitoring, and active probing, to both understand path anomalies and to provide optimization opportunities for the host service.

1 Introduction

As the Internet grows and routing complexity increases, network-level instabilities are becoming more common. Among the problems causing end-to-end path failures are router misconfigurations [16], maintenance, power outages, and fiber cuts [15]. Inter-domain routers may take tens of minutes to reach a consistent view of the network topology after network failures, during which time end-to-end paths may experience outages, packet losses and delays [14]. These routing problems can affect performance and availability [1, 14], especially if they occur on commonly-used network paths. However, even determining the existence of such problems is nontrivial, since

no central authority monitors all Internet paths.

Previously, researchers have used routing messages, such as BGP [16], OSPF [15] and IS-IS [12] update messages to identify inter-domain and intra-domain routing failures. This approach usually requires collecting routing updates from multiple vantage points, which may not be easily accessible for normal end users. Other researchers have relied on some form of distributed active probing, such as pings and traceroutes [1, 7, 19], to detect path anomalies from end hosts. These approaches monitor the paths between pairs of hosts by having them repeatedly probe each other. Because this approach requires cooperation from both source and destination hosts, these techniques measure only paths among a limited set of participating nodes.

We observe that there exist several *wide-area services* employing multiple geographically-distributed nodes to serve a large and dispersed client population. Examples of such services include Content Distribution Networks (CDNs), where the clients are distinct from the nodes providing the service, and Peer-to-Peer (P2P) systems, where the clients also participate in providing the service. In these kinds of systems, the large number of clients use a variety of network paths to communicate with the service, and are therefore likely to see any path instabilities that occur between them and the service nodes.

This scenario of geographically-distributed clients accessing a wide-area service can itself be used as a monitoring infrastructure, since the natural traffic generated by the service can reveal information about the network paths being used. By observing this traffic, we can passively detect odd behavior and then actively probe it to understand it in more detail. This approach produces less overhead than a purely active-probing based approach.

This monitoring can also provide direct benefit to the wide-area service hosting the measurement infrastructure. By characterizing failures, the wide-area service can mitigate their impact. For example, if the outbound path between a service node and a client suddenly fails,

it may be possible to mask the failure by sending outbound traffic indirectly through an unaffected service node, using techniques such as overlay routing [1]. More flexible services may adapt their routing decisions, and have clients use service nodes that avoid the failure entirely. Finally, a history of failure may motivate placement decisions—a service may opt to place a service node within an ISP if intra-ISP paths are more reliable than paths between it and other ISPs.

This paper describes a monitoring system, PlanetSeer, that has been running on PlanetLab since February 2004. It passively monitors traffic between PlanetLab and thousands of clients to detect anomalous behavior, and then coordinates active probes from many PlanetLab sites to confirm the anomaly, characterize it, and determine its scope. We are able to confirm roughly 90,000 anomalies per month using this approach, which exceeds the rate of previous active-probing measurements by more than two orders of magnitude [7]. Furthermore, since we can monitor traffic initiated by clients outside PlanetLab, we are also able to detect anomalies beyond those seen by a purely active-probing approach.

In describing PlanetSeer, this paper makes three contributions. First, it describes the design of the passive and active monitoring techniques we employ, and presents the algorithms we use to analyze the failure information we collect. Second, it reports the results of running PlanetSeer over a three month period of time, including a characterization of the failures we see. Third, it discusses opportunities to exploit PlanetSeer diagnostics to improve the level of service received by end users.

2 Background

Although the Internet is designed to be self-healing, various problems can arise in the protocols, implementations [15], and configurations [16] to cause network instability. Even in the absence of such problems, routing updates can take time to propagate, so failures may be visible for minutes rather than seconds. Even though tools like *ping* and *traceroute* exist for diagnosing network problems, pinpointing failures and determining their origins is nontrivial for several reasons:

Network paths are often asymmetric. Paxson observed that 49% of node pairs have different forward and reverse paths which visit at least one different city [19]. Since *traceroute* only maps the forward path, it is hard to infer whether the forward or reverse path is at fault without cooperation from the destination. PlanetSeer combines passive monitoring results, path history information, and multi-point probing to isolate forward failures.

Failure origin may differ from failure appearance. Routing protocols, such as BGP and OSPF, may propa-

gate failure information to divert traffic away from failed links. When a traceroute stops at a hop, it is often the case that the router has received a routing update to withdraw that path, leaving no route to the destination.

Failure durations are highly varied. Some failures, like routing loops, can last for days. Others may persist for less than a minute. This high variance makes it hard to diagnose failures and react in time.

Failure isolation requires broad coverage. Historically, few sites had enough network coverage to initiate enough traceroutes to identify all affected paths. The advent of public traceroute servers [24] provides some resources to help manually diagnose problems, and tools such as ScriptRoute [22] can help automate the process.

The advent of wide-coverage networking testbeds like PlanetLab has made it possible to deploy wide-area services on a platform with enough network coverage to also perform the probing on the same system. This approach addresses the problems listed above: (1) when clients initiate connections to the wide-area service, we obtain a forward path that we can monitor for anomalies; (2) PlanetLab nodes span a large number of diverse autonomous systems (ASes), providing reasonable network coverage to initiate probing; and (3) active probing can be launched as soon as problems are visible in the passively-monitored traffic, making it possible to catch even short-term anomalies that last only a few minutes.

While our focus is techniques for efficiently identifying and characterizing network anomalies, we must give some attention to the possibility of our host platform affecting our results. In particular, it has been recently observed that intra-PlanetLab paths may not be representative of the Internet [2], since these nodes are often hosted on research-oriented networks. Fortunately, by monitoring services with large client populations, we conveniently bypass this issue since most of the paths being monitored terminate outside of PlanetLab. By using geographically-dispersed clients connecting to a large number of PlanetLab nodes, we observe more than just intra-PlanetLab connections.

3 PlanetSeer Operation

This section describes our environment and our approach, including how we monitor traffic, identify potential path anomalies, and actively probe them.

3.1 Components

We currently use the CoDeeN Content Distribution Network [26] as our host wide-area service, since it attracts a reasonably large number of clients (7K-12K daily) and generates a reasonable traffic volume (100-200 GB/day, 5-7 million reqs/day). CoDeeN currently runs on 120

PlanetLab nodes in North America (out of 350 worldwide), but it attracts clients from around the world. CoDeeN acts as a large, cooperative cache, and it forwards requests between nodes. When it does not have a document cached, it gets the document from the content provider (also known as the origin server). As a result, in addition to the paths between CoDeeN and the clients, we also see intra-CoDeeN paths, and paths between CoDeeN and the origin servers.

PlanetSeer consists of a set of passive monitoring daemons (MonD) and active probing daemons (ProbeD). The MonDs run on all CoDeeN nodes, and watch for anomalous behavior in TCP traffic. The ProbeDs run on all PlanetLab nodes, including the CoDeeN nodes, and wait to be activated. When a MonD detects a possible anomaly, it sends a request to its local ProbeD. The local ProbeD then contacts ProbeDs on the other nodes to begin a coordinated planet-wide probe. The ProbeDs are organized into groups so that not all ProbeDs are involved in every distributed probe.

Currently, some aspects of PlanetSeer are manually configured, including the selection of nodes and the organization of ProbeD groups. Given the level of trust necessary to monitor traffic, we have not invested any effort to make the system self-organizing or open to untrusted nodes. While we believe that both goals may be possible, these are not the current focus of our research.

Note that none of our infrastructure is CoDeeN-specific or PlanetLab-specific, and we could easily monitor other services on other platforms. For PlanetSeer, the appeal of CoDeeN (and hence, PlanetLab) is its large and active client population. The only requirement we have is the ability to view packet headers for TCP traffic, and the ability to launch traceroutes. On PlanetLab, the use of *safe raw sockets* [3] mitigates some privacy issues – the PlanetSeer service only sees those packets that its hosting service (CoDeeN) generates. In other environments, we believe the use of superuser-configured in-kernel packet filters can achieve a similar effect.

In terms of resources, neither ProbeD nor MonD require much memory or CPU to run. The non-glibc portion of ProbeD has a 1MB memory footprint. The MonD processes have a memory consumption tied to the level of traffic activity, and is used to store flow tables, statistics, etc. In practice, we find that it requires roughly 1KB per simultaneous flow, but we have made no effort to optimize this consumption. The CPU usage of monitoring and probing is low, with only analysis requiring much CPU. Currently, analysis is done offline in a centralized location, but only so we can reliably archive the data. We could perform the analysis on-line if desired – currently, each anomaly requires a 20 second history to detect, one minute to issue and collect the probes, and less than 10ms of CPU time to analyze.

3.2 MonD Mechanics

MonD runs on all CoDeeN nodes and observes all incoming/outgoing TCP packets on each node using PlanetLab's *tcpdump* utility. It uses this information to generate path-level and flow-level statistics, which are then used for identifying possible anomalies in real-time.

Although flow-level information regarding TCP timeouts, retransmissions, and round-trip times (RTTs) already exists inside the kernel, this information is not easily exported by most operating systems. Since MonD runs as a user-level process, it instead derives this information by observing packet-level activity from *tcpdump*. It instead infers flow-level information—e.g., timeouts, retransmissions, and round trip times (RTTs)—from the sniffed packets, and aggregates information from flows on the same path to infer anomalies on that path.

MonD maintains path-level and flow-level information, with paths identified by their source and destination IP addresses, and flows identified by both port numbers in addition to the addresses. Flow-level information includes information such as sequence numbers, timeouts, retransmissions, and round-trip times. Path-level information aggregates some flow-level information, such as loss rates and RTTs.

MonD adds new entries when it sees new paths/flows. On packet arrival, MonD updates a timestamp for the flow entry. Inactive flows, which have not received any traffic in *FlowLifeTime* (15 minutes in the current system), are pruned from the path entry, and any empty paths are removed from the table.

3.3 MonD Behavior

MonD uses two indicators to identify possible anomalies, which are then forwarded to ProbeD for confirmation. The first indicator is a change in a flow's Time-To-Live (TTL) field. The TTL field in an IP packet is initialized by a remote host and gets decremented by one at each hop along the traversed path. If the path between a source and destination is static, the TTL value of all packets that reach the destination should be the same. If the TTL changes in the middle of the stream, it usually means a routing change has occurred.

The second indicator, multiple consecutive timeouts, signals a possible path anomaly since such timeouts should be relatively rare. A TCP flow can time out several times from a single unacknowledged data packet, and each consecutive timeout causes the retransmission timeout period to double [25]. The minimum initial retransmission timeout in TCP ranges from 200ms (in Linux) to 1 second (in RFC 2988 [25]). Thus, n consecutive timeouts means either the data packets or the corresponding acknowledgment packets (ACKs) have not been received during the last $2^n - 1$ periods (seconds or 200ms ticks).

Our current threshold is four consecutive timeouts, which corresponds to 3.2–16 seconds. Since most congestion periods on today's Internet are short-lived (95% are less than 220ms [27]), these consecutive timeouts are likely due to path anomalies. We can further subdivide this case based on whether MonD is on the sender or receiver side of the flow. If MonD is on the receiver side, then no ACKs are reaching the sender, and we can infer the problem is on the path from the CoDeeN node to the client/server, which we call **forward path**. If MonD is on the sender side, then we cannot determine whether outbound packets are being lost or whether ACKs are being lost on the way to MonD.

3.4 MonD Flow/Path Statistics

We now describe how MonD infers path anomalies after grouping packets into flows. We examine how to measure the per-flow RTTs, timeouts and retransmissions.

To detect timeouts when MonD is on the sender side, we maintain two variables, *SendSeqNo* and *SendRtxCount* for each flow. *SendSeqNo* is the sequence number (seqno) of the most recently sent new packet, while *SendRtxCount* is a count of how many times the packet has been retransmitted. If we use *CurrSeqNo* to represent the seqno of the packet currently being sent, we see three cases: If *CurrSeqNo* > *SendSeqNo*, the flow is making progress, so we clear *SendRtxCount* and set *SendSeqNo* to *CurrSeqNo*. If *CurrSeqNo* < *SendSeqNo*, the packet is a fast retransmit, and we again set *SendSeqNo* to *CurrSeqNo*. If *CurrSeqNo* = *SendSeqNo*, we conclude a timeout has occurred and we increment *SendRtxCount*. If *SendRtxCount* exceeds our threshold, MonD notifies ProbeD that a possible path anomaly has occurred.

A similar mechanism is used when MonD observes the receiver side of a TCP connection. It keeps track of the largest seqno received per flow, and if the current packet has the same seqno, a counter is incremented. Doing this determines how many times a packet has been retransmitted due to consecutive timeouts at the sender. When this counter hits our threshold, MonD notifies ProbeD that this sender is not seeing our ACKs. Since we are seeing the sender's packets, we know that this direction is working correctly. Note that this method assumes that duplicate packets are mostly due to retransmissions at the sender. This assumption is safe because previous work has shown that packets are rarely duplicated by the network [20].

Detecting TTL change is trivial: MonD records the TTL for the first packet received along each path. For each packet received from any flow on the same path, we compare its TTL to our recorded value. If MonD detects any change, it notifies ProbeD that a possible anomaly has occurred. Note that this case can aggregate information from all flows along the path.

3.5 ProbeD Operation

ProbeD is responsible for the active probing portion of PlanetSeer, and generally operates after being notified of possible anomalies by MonD. For the purpose of the following discussion, when an anomaly occurs, we call the CoDeeN node where the anomaly is detected the *local node*, and the corresponding remote client/server the *destination*. The active probing is performed using traceroute, a standard network diagnostic tool. ProbeD supports three probing operations:

Baseline Probes: When a new IP address is added to MonD's path table, the ProbeD on the local node performs a "baseline probe" to that destination. It is expected that the results of this traceroute reflect the default network path used to communicate with that destination under normal conditions. For actively-used communication paths, a baseline probe is launched once every 30 minutes. When PlanetSeer is run on CoDeeN nodes, these baseline probes are generated whenever a new client connects to a node, or when a node has to contact a new origin server.

Forward Probes: When a possible anomaly is detected by the local MonD and reported to ProbeD, it invokes multiple traceroutes from a set of geographically distributed nodes (including itself) to the destination; we call the traceroute from the local node the *local traceroute* or *local path*. This process is performed twice, generally within one minute, in order to identify what we term *ultrashort* anomalies. On particularly problematic paths, MonD might report possible anomalies very frequently, especially if the path is very unstable. To avoid generating too much probing traffic, ProbeD rate-limits the forward probes so that it does not probe the same destination within 10 minutes.

Reprobes: If the forward probes confirm an anomaly along a path to a destination, the local ProbeD that initiated the forward probes periodically reprobes that path to determine the duration and effects of the anomaly. We currently reprobe four times, at 0.5, 1.5, 3.5, and 7.5 hours after the anomaly detection time. These reprobes can compare their traceroute results with the original baseline probe as well as the forward probes.

3.6 ProbeD Mechanics

When ProbeD performs the forward probes, it launches them from geographically distributed nodes on PlanetLab. Compared with only doing traceroute from the local node, using multiple vantage points gives us a more

Category	Grps	Sites	Descriptions
US (edu)	11	70	US Universities
US (non-edu)	5	13	Intel, HP, NEC, etc.
Canada	2	11	Eastern & Western Canada
Europe	7	31	UK, France, Germany, etc.
Asia & MidE	4	14	China, Korea, Israel, etc.
Others	1	6	Australia, Brazil, etc.
Total	30	145	

Table 1: Groups of the probing sites

complete view of the anomaly, such as its location, pattern, and scope. Our ProbeDs are running on 353 nodes across 145 sites on PlanetLab, more than the number of nodes running CoDeeN. They are distributed across North/South America, Europe, Asia and elsewhere.

Since the set of ProbeDs must communicate with each other, they keep some membership information to track liveness. We note that an unavailable ProbeD only results in a degradation of information quality, rather than complete inoperability, so we do not need to aggressively track ProbeD health. Each ProbeD queries the others when it first starts. Thereafter, dead ProbeDs are checked every 8 hours to reduce unneeded communication. In the course of operation, any ProbeD that is unresponsive to a query is considered dead.

We divide the ProbeD nodes into 30 groups based on geographic diversity, as shown in Table 1, mainly to reduce the number of probes launched per anomaly. Probing every anomaly from every ProbeD location would yield too much traffic (especially to sites with conservative intrusion detection systems), and the extra traffic may not yield much insight if many nodes share the same paths to the anomaly. We also divide North America into educational and non-educational groups, because the educational (.edu) sites are mostly connected to Internet2, while non-educational sites are mostly connected by commercial ISPs.

When a ProbeD receives a request from its local MonD, it forwards it to a ProbeD in each of the other groups. The ProbeDs perform the forward probes, and send the results back to the requester. All results are collected by the originator, and logged with other details, such as remote IP address and the current time.

3.7 Path Diversity

We have been running PlanetSeer since February 2004. In three months, we have seen 887,521 unique clients IPs, coming from 9232 Autonomous Systems (ASes) (according to previous IP-to-AS mappings techniques [17]). Our probes have traversed 10090 ASes, well over half of all ASes on the Internet. We use a hierarchical AS classification scheme that has five tiers, based on AS relationships and connectivity [23]. The highest layer (tier 1) represents the *dense core* of the In-

Tier #	Covered	Tier Size	Coverage
Tier 1	22	22	100%
Tier 2	207	215	96%
Tier 3	1119	1392	80%
Tier 4	1209	1420	85%
Tier 5	5906	13872	43%
Unmapped	1627		

Table 2: Path diversity

ternet, and consists of 22 ASes of the largest ISPs. Tier 2 typically includes ASes of other large national ISPs. Tier 3 includes ASes of regional ISPs. Finally, tiers 4 and 5 include ASes of small regional ISPs and customer ASes respectively. As shown in Table 2, we have very good coverage of the top 4 AS tiers, with complete coverage of tier 1 and nearly-complete coverage of tier 2.

4 Confirming Anomalies

Having collected the passive data from MonD and the traceroutes from ProbeD, the next step is processing the probes to confirm the existence of the anomaly. This section describes how we use this data to classify anomalies and quantify their scope. It also reports how different types of anomalies influence end-to-end performance.

4.1 Massaging Traceroute Data

Some of the data we receive from the traceroutes is incomplete or unusable, but we can often perform some simple processing on it to salvage it. The unusable hops in a traceroute are those that do not identify the routers or that identify the routers by special-use IP addressess [11]. The missing data is generally the absence of a hop, and can be interpolated from other traceroutes.

Identifying and pruning unusable hops in traceroute is simple: the unusable hops are identified by asterisks in place of names, and other than showing the existence of these hops, these data items provide no useful information. We simply remove them from the traceroute but keep the relative hop count difference between the existing hops.

Missing hops in a traceroute are slightly harder to detect, but we can use overlapping portions of multiple traceroutes to infer where they occur. We use a simple heuristic to identify the missing hops: we compare traceroutes that share the same destination, and if the hops leading to the destination differ by an intermediate hop that is present in one and missing in the other, we replace the missing hop with the address in the other trace. Put more formally, given two traceroutes from two sources to the same destination, suppose there are two subsequences of these two traceroutes, $X(X_1, X_2, \dots, X_m)$ and $Y(Y_1, Y_2, \dots, Y_n)$ ($m > 2$ and $n > 2$) such that $X_1 = Y_1$ and $X_m = Y_n$. We

TTL Change	Timeout	Fwd Timeout
994485 (44%)	754434 (33%)	510669 (23%)

Table 3: Breakdown of anomalies reported by MonD

define $hop(X_i)$ to be the hop count of X_i . Note that the number of hops between X_1 and X_m , $hop(X_m) - hop(X_1)$, can be greater than $m - 1$, because we have removed “*” and special-use IPs from the traceroutes. If $hop(X_m) - hop(X_1) = hop(Y_n) - hop(Y_1)$, it is very likely that all the hops in X and Y are the same since they merge at $X_1(Y_1)$ and follow the same number of hops to $X_m(Y_n)$. If there exists X_i such that the hop corresponds to $hop(Y_1) + hop(X_i) - hop(X_1)$ in Y does not exist because it has been removed as “*”, we consider X_i a missing hop in Y and add this hop into Y .

Our approach to inserting missing hops helps us filter out the “noise” in traceroutes so that it does not confuse our anomaly confirmation using route change as described in Section 4.2. However, it may mask certain hop differences. For example, we sometimes see two paths X and Y merge at $X_1(Y_1)$, and diverge at some later hops before merging at $X_m(Y_n)$ again. This usually occurs between tightly-coupled routers for load balancing reasons, where a router selects the next hop from several parallel links based on the packet IP addresses and/or traffic load [19]. In this case, inserting missing hops may eliminate the different hops between the two traceroutes.

For our purposes, we do not treat such “fluttering” [19] as anomalies because it occurs as a normal practice. We detect fluttering by looking for hops X_i and Y_j such that $hop(Y_j) - hop(Y_1) = hop(X_i) - hop(X_1)$ but $X_i \neq Y_j$, and we merge them into the same hop in all the traceroutes. Note that this could also possibly eliminate the hop difference caused by path change and lead to underestimating the number of anomalies.

4.2 Final Confirmation

After we have processed the traceroutes, we are ready to decide whether an event reported by MonD is actually an anomaly. We consider an anomaly “confirmed” if *any* of the following conditions is met:

Loops: There is a loop in the local traceroute from the local node to the destination, which means the anomaly is triggered by routing loops.

Route change: The local traceroute disagrees with the baseline traceroute. Note that the baseline traceroute is no more than 30 minutes old. Given that 91% of the Internet paths remains stable for more than several hours [19], the anomaly is most likely caused by path change or path outage.

Non-Anomaly	Anomaly	Undecided
1484518 (66%)	271898 (12%)	503172 (22%)

Table 4: Breakdown of reported anomalies using the four confirmation conditions

Partial unreachability: The local traceroute stops before reaching the destination, but there exist traceroutes from other nodes that reach the destination. This could be caused by path outages.

Forwarding failures: The local traceroute returns an ICMP destination unreachable message, with code of net unreachable, host unreachable, net unknown, or host unknown. This usually indicates that a router does not know how to reach the destination because of routing failures [13].

Our confirmation process is very conservative—it is possible that some of the reported anomalies are real, but do not meet any of the above conditions. However, our goal is to obtain enough samples of anomalies for our analysis and we do not want our results to be tainted by false positives. Hence, we choose stricter conditions for confirming anomalies. Similarly, we confirm a reported anomaly as *non-anomaly* if the local traceroute does not contain any loop, agrees with the baseline traceroute, and reaches the destination. For a confirmed non-anomaly, we do not perform traceroutes at remote ProbeDs, in order to reduce measurement traffic.

In three months, we have seen a total of 2,259,588 possible anomalies reported, of which we confirmed 271,898. Table 3 shows the number of reported anomalies of each type. As we can see, TTL change is the most common type of reported anomaly, accounting for 44% of the reported anomalies. For the remaining anomalies triggered by timeouts, passive monitoring suggests that 23% are most likely caused by forward path problems.

Table 4 shows the breakdown of anomalies using the 4 confirmation conditions. The non-anomalies account for 2/3 of the reported anomalies. Among the possible reasons for the occurrence of non-anomalies are: ultra-short anomalies, path-based TTL, and aggressive consecutive timeout levels. Some anomalies, which we term ultrashort, are so short that our system is unable to respond in time. Since they often are in the process of being resolved when the forward probes are taking place, the traceroute results may be inconsistent. Many false alarms from path-based TTL changes are due to NAT boxes. When clients with different initial TTL values share a NAT box, their interleaved packets appear to show TTL change. Using flow-based TTL change would reduce these false alarms, but may miss real TTL changes that occur between flows since any path history

	Temporary	Persistent
Total	3565 (17%)	18000 (83%)
$\leq 30\text{min}$	N/A	54%
$\leq 1.5\text{ hrs}$	N/A	11%
$\leq 3.5\text{ hrs}$	N/A	6%
$\leq 7.5\text{ hrs}$	N/A	6%
$> 7.5\text{ hrs}$	N/A	23%
Single Loop	3008 (84%)	17007(94%)
Multiple Loops	557 (16%)	993 (6%)
1 AS	3021, (85%)	17895, (99%)
2 ASes	416, (12%)	101, (1%)
3 ASes	106, (3%)	4, (0%)
$\geq 4\text{ ASes}$	22, (0%)	0, (0%)
Tier-1 AS	510 (15%)	244 (2%)
Tier-2 AS	859 (25%)	789 (6%)
Tier-3 AS	1378(40%)	6263 (46%)
Tier-4 AS	197 (5%)	3899 (29%)
Tier-5 AS	538 (15%)	2401 (17%)
Total	3482	13596

Table 5: Summarized breakdown of 21565 loop anomalies. Some counts less than 100% because some ASes are not in the AS hierarchy mapping.

would be lost. Finally, our consecutive timeout conditions may be aggressive for hosts with short timeout periods. Excluding the non-anomalies, we confirm 271898 (35%) of the remaining anomalies and probe them from multiple vantage points. We use these anomalies for our analysis in the remainder of this paper.

5 Loop-Based Anomalies

This section focuses on analyzing routing loops, which can occur due to inconsistencies in routing state, misconfiguration, and other causes. We are interested in their frequency, duration, size, location, and effect on end-to-end performance.

We detect routing loops by observing the same sequence of routers appearing several times in a traceroute. Since loops in traceroute may reflect upstream routing changes rather than true routing loops [19], we choose to take a conservative approach and require that the same sequence appear in a traceroute **at least 3 times** before we confirm it as a routing loop.

Using this metric, we identify a total number of 21565 routing loops in our data. If we relax the loop condition to allow loops that have the same sequence only twice, this count increases to 119,936, almost six times as many. Using this approach, our overall confirmed anomaly count would increase 36% to over 370K.

Loops are separated into persistent and temporary loops [19] based on whether the traceroute was ultimately able to exit the loop. If the traceroute stays within

the loop until the maximum number of hops (32 in our case), we classify it as persistent, while if the loop is resolved before the traceroute reaches the maximum hop, it is temporary. Temporary loops can occur due to the time necessary to propagate updated routing information to the different parts of the network, while persistent loops can be caused by various reasons, including router misconfiguration [16]. Persistent loops tend to last longer and may require human intervention to be resolved, so they are considered more harmful. About 83% of the observed loops are persistent, as shown in Table 5. Since temporary loops only exist for a short period, it may be harder to catch them.

We use the reprobes to determine duration of the persistent loops. The reprobes for some persistent loops are missing, often because the local PlanetLab node failed or rebooted before all reprobes completed. For those loops, we do not know when the loops were resolved. We only include the loops having all 4 reprobes in our analysis. Therefore, we show the percentage of loops in each duration instead of the exact numbers in Table 5. We can see many persistent loops are either resolved quickly (54% terminate within half an hour) or last for a long time (23% stay for more than 7.5 hours).

Previous research has noted that routing loops are likely to be correlated [19], because nearby routers usually share routing information very quickly. If some routers have inconsistent information, such information is likely to be propagated to other nearby routers and cause those routers to form loops. We observe a similar phenomenon, which is quantified in Table 5. We count the number of distinct loops in traceroutes from other ProbeDs during the same period. We find that 16% of the temporary loops are accompanied by at least one disjoint loop while only 6% of the persistent loops see them. We suspect the reason is temporary loops are more likely to stem from inconsistent routing state while persistent loops are more likely to be caused by other factors which may not be related to routing inconsistency.

Finally, using the persistent loop data, we can also get some insight into the relative distribution of AS quality by measuring how evenly-distributed the loops are. Since these loops are largely single-AS, they are very unlikely to arise from external factors, and may provide some insight into the monitoring/management quality of the AS operators. We use a metric, which we call *skew*, to provide some insight into the distribution. We calculate skew as the percentage of per-tier loops seen by the “worst” 10% of the ASes in that tier. A skew value of 10% indicates all ASes in the tier are likely to be uniform in the quality, while larger numbers indicate a wider disparity between the best ASes and the worst.

In tier 1, the top 2 ASes (10% of 22) account for 35% of the loops, while in tier 2, the top 21 ASes (10% of 215)

	2	3	4	5	6+
Persistent/All	97%	2%	1%	0%	0%
Persistent/Core	94%	4%	1%	1%	0%
Persistent/Edge	97%	2%	1%	0%	0%
Temporary/All	51%	29%	11%	7%	2%
Temporary/Core	45%	31%	13%	8%	3%
Temporary/Edge	53%	27%	12%	6%	2%

Table 6: Number of hops in loops, as % of loops

account for 97% of the loops. This skew may translate into different reliabilities for the customers they serve. The disparity in traffic must also be considered when judging how important these skew numbers are. With respect to the traffic that we observe, we find that these ASes account for 20% of tier 1 traffic and 63% of tier 2 traffic. The disparity between the loop rates and the traffic for these ASes would indicate that these ASes appear to be much more problematic than others in their tier.

5.1 Scope

Besides their frequency, one of the other factors that determines the effect of routing loops is their *scope*, including how many routers/ASes are involved in the loop, and where they are located. We use the term *loop length* to refer to the number of routers involved, and we show a breakdown of this metric in Table 6. The most noticeable feature is that temporary loops have much longer lengths than persistent loops. 97% of the persistent loops consist of only 2 routers, while the ratio is only 50% for temporary loops. Intuitively, the more routers are involved in a loop, the less stable it is. Therefore, most persistent loops exist between only two routers, while temporary loops span additional routers as the inconsistent state propagates.

We next examine the number of ASes that are involved in the loops. The breakdown is shown in Table 5, which shows that persistent loops overwhelmingly occur within a single AS, while 15% of temporary loops span multiple ASes. Ideally, BGP prevents any inter-AS loops by prohibiting a router from accepting an AS path with its own AS number in that path. However, BGP allows transient inconsistency, which can arise during route withdrawals and announcements [10], and this is why we see more temporary loops spanning multiple ASes. In contrast, persistent loops can occur due to static routing [19] or router misconfigurations [16]. Given how few persistent loops span multiple ASes, it appears that BGP's loop suppression mechanism is effective.

To understand where loops arise, we classify them according to the hierarchy of ASes involved. In theory, we could calculate their depth [7], which would tell us the minimum number of hops from the routers to the network edge. However, since we cannot launch probes

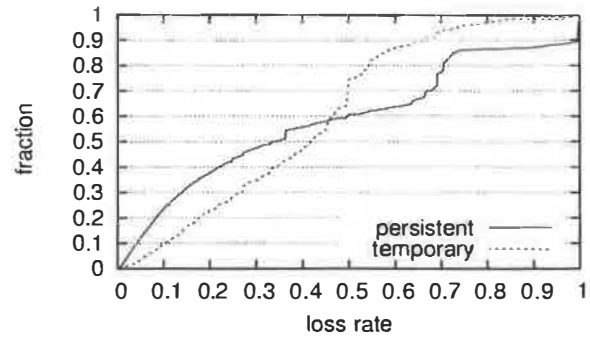


Figure 1: CDF of loss rates preceding the loop anomalies

from the clients, this depth metric would be misleading. If the loop occurs on an AS that does not lie near any of our ProbeD locations, our probes travel through the network core to reach it, and we would believe it to be very far from the edge. If we could launch probes from the clients, network depth would be meaningful.

We map loops into tiers by using the tier(s) of the AS(es) involved. A loop can be mapped to multiple tiers if it involves multiple ASes. Table 5 shows the number of loops occurring in each tier. Tier-1 and tier-2 ASes have very few persistent loops, possibly because they are better provisioned than smaller ASes. A large portion of loops (40% for temporary and 46% for persistent) occur in tier-3 (outer core) ASes, which suggests that the paths in those large regional ASes are less stable. In Table 6, we compare the loops in the core network (tiers 1, 2, 3) or the edge network (tiers 4, 5). As the table shows, both temporary and persistent loops are likely to involve more hops if occurring in the core network.

5.2 End-to-End Effects

Loops can degrade end-to-end performance in two ways: by overloading routers due to processing the same packet multiple times [10] (for temporary loops), or by leading to loss of connectivity between pairs of hosts (for permanent loops). Since MonD monitors all flows between the node and remote hosts, we can use the network statistics it keeps to understand end-to-end effects.

When MonD suspects an anomaly, it logs the retransmission rate and RTT *leading up to that point*. Retransmission rates are calculated for the last 5 minutes. RTTs are calculated using an Exponentially Weighted Moving Average (EWMA) with the most recent value given a weight of 1/8, similar to that used in TCP. Figure 1 shows the CDF of the retransmission rate, and we see that 65% of the temporary loops and 55% of the persistent loops are preceded by loss rates exceeding 30%. Since the typical Internet loss rate is less than 5% [20], this higher loss rate will significantly reduce end-user TCP performance prior to the start of the anomaly.

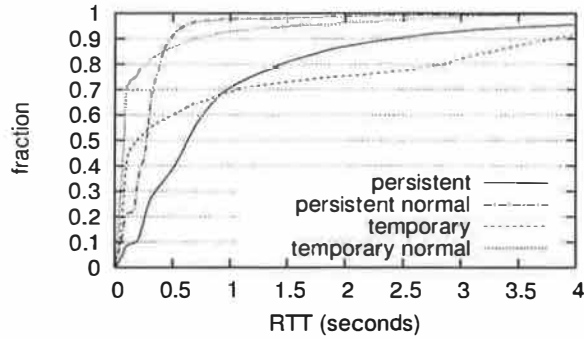


Figure 2: CDF of RTTs preceding the loop anomalies vs. under normal conditions

In addition to the high loss rates, loop anomalies are also preceded by high latency, as shown in Figure 2. High latency can be caused by queuing due to congestion or packets repeatedly traversing the same sequence of routers. We compare the RTT right before loops occur with the RTT measured in the baseline traceroute on the same path. It is evident that loops can significantly degrade RTTs.

6 Building a Reference Path

While loop-based anomalies are relatively simple to identify, they represent a relatively small fraction of the total confirmed anomalies. Classifying the other anomalies, however, requires more effort. This section discusses the steps taken to classify the non-loop anomalies, and the complications involved. The main problem is how to determine that the anomaly occurs on the forward path from the local node to the destination. Additionally, we want to characterize the anomaly's features, such as its pattern, location and affected routers.

To deal with these two problems, we need a *reference path* from the local node to the destination, which represents the path before the anomaly occurs. Then we can compare it against the local traceroute during the anomaly. This comparison serves three purposes: First, it can help us distinguish between forward-path and reverse-path anomalies. If the local path during the anomaly is different from the reference path, the route change usually indicates that the anomaly is on the forward path. Second, it can be used to quantify the scope of the anomaly. By examining which routers overlap between the local path and the reference path, we can estimate which routers are potentially affected by the anomaly. It can also be used to determine the location of the anomaly, in terms of the number of the router hops between the anomaly and the end hosts. Third, it is used to classify the anomalies. Based on whether the local traceroute reaches the last hop of the reference path, we can classify it as either path change or path outage.

Ideally, we could simply use the baseline traceroute as the reference path, if it successfully reaches the destination. If the local traceroute during an anomaly stops at some intermediate hop, we know it is an outage. If the local traceroute is different, but reaches the destination, we know it is a routing change. However, the baseline traceroute may not reach the destination for a variety of reasons. Some of these include:

- The destination is behind a firewall which filters traceroutes. In this case, we still want to use it as the reference path, which can be compared with local traceroute to analyze anomalies.
- Some intermediate routers filter traceroutes. In this case, we do not have enough information about the hops after the last known hop on the forward path. When an outage occurs, we cannot quantify where it occurs since the anomaly may occur after the last known hop.
- The baseline traceroute is also affected by the anomaly and fails to reach the destination. In this case, we cannot use it as a reference because it usually does not provide more useful information than the local traceroute.

The rest of this section focuses on deciding whether the baseline traceroute can be used as the reference path when it does not reach the destination. If a baseline path S stops at hop S_x , we try to guess if S_x is a firewall using some heuristics. S_x must meet the following four requirements before we consider it a firewall:

1. From MonD's passive data, we know the client is able to send and receive TCP packets with the local node. Therefore, the path is working when the baseline traceroute is being calculated.
2. S does not return an ICMP destination unreachable message, which usually indicates that the traceroute encounters routing problems at S_x [13].
3. S_x and the destination are in the same AS. We assume that a firewall and its protected clients should belong to the same organization.
4. S_x is within n hops (close) to the destination.

The first three requirements are easy to verify, so we focus on the last requirement. Let $RevHop(h)$ be the number of hops from hop h to the local node on the reverse path. We first want to check if $0 < RevHop(dst) - RevHop(S_x) \leq n$. From MonD, we know $RevTTL(dst)$, the TTL of a packet when it arrives at the local node from the destination. If

Type	Number	Percentage
Path Change	120,283	48%
Forward Outage	23,921	10%
Other Outage	62,107	24%
Temporary	44,022	18%
Total	250,333	100%

Table 7: Non-loop anomalies breakdown

the TTL is initialized to $InitTTL(dst)$ by the destination, we have $InitTTL(dst) - RevTTL(dst) = RevHop(dst)$ because the TTL is decremented at each hop along the reverse path. The issue is how to determine $InitTTL(dst)$. The initial TTL values differ by OS, but are generally one of the following: 32 (Win 95/98/ME), 64 (Linux, Tru64), 128 (Win NT/2000/XP), or 255 (Solaris). Because most Internet paths have less than 32 hops, we can simply try these 4 possible initial TTL values and see which one, when subtracted by $RevTTL(dst)$, gives a $RevHop(dst)$ that is less than 32 hops. We will use that as $InitTTL(dst)$ to calculate $RevHop(dst)$. Similarly, from the traceroute, we can also calculate $RevHop(S_x)$ using $RevTTL(S_x)$.

Although inter-AS routing can be asymmetric, intra-AS paths are usually symmetric [19]. Since S_x and the destination are in the same AS, their forward hop count difference should be the same as their reverse hop count difference, which we are able to compute as described above.

Choosing an appropriate n for all settings is difficult, as there may be one or more hops between a firewall and its protected hosts. We conservatively choose $n = 1$, which means we consider S as a valid reference path only when S_x is one hop away from the destination. This will minimize the possibility that a real path outage is interpreted as a traceroute being blocked by a firewall. However, it leads to bias against large organizations, where end hosts are multiple hops behind a firewall. In such cases, we cannot determine if the anomalies are due to path outage or blocking at a firewall. Therefore, we do not further analyze these anomalies.

7 Classifying Non-loop Anomalies

In this section, we discuss classifying anomalies by comparing the reference path R with the local path L . There are three possibilities when we compare L and R :

1. L reaches the last hop of R . In this case, L must differ from R in some intermediate hops, or else we would not have confirmed it as an anomaly. This case corresponds to a *path change*, which will be discussed in Section 7.1.
2. If L stops at some intermediate hop of R , it could

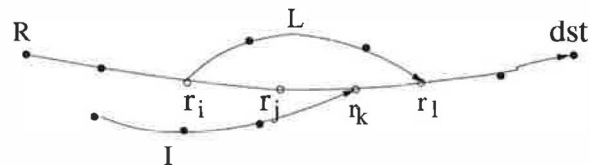


Figure 3: Confining the scope of path change

be due to path outage on the forward path or reverse path failure. We will describe how to distinguish between them in Section 7.2.

3. If L diverges from R after some hops and stops before merging into R , we consider it as a path outage although it is accompanied by a route change. We will also discuss this case in Section 7.2.

We observe a total of 250333 non-loop anomalies, with their breakdown shown in Table 7. About half of them are path changes, and 10% are forward path outages. For the 24% that are classified as other outages, we cannot infer whether they are on the forward or reverse paths. The remaining 18% are temporary anomalies. In these cases, the first local traceroute does not match the reference path, but the second local traceroute matches. In these cases, the recovery has taken place before we can gather the results of the remote probes, making characterization impossible. While it is possible that some remote probes may see the anomaly, the rapidly-changing state is sure to cause inconsistencies if we were to analyze them. To be conservative, we do not perform any further analysis of these anomalies, and focus only on path changes and forward outages. These temporary anomalies are different from the ultrashort anomalies in that the ultrashort anomalies were already in the repair process during the first probe. So, while we choose not to analyze temporary anomalies further, we can at least inarguably confirm their existence, which is not the case with the ultrashort anomalies.

7.1 Path Changes

We first consider path changes, in which the local path L diverges from the reference path R after some hops, then merges back into R and successfully terminates at the last hop of R . This kind of anomaly is shown in Figure 3.

7.1.1 Scope and End-to-End Effects

As discussed in Section 2, it is usually very difficult to locate the origin of path anomalies purely from end-to-end measurement [8]. However, even if the precise origin cannot be determined, we may be able to narrow the *scope* of the anomaly. We define the scope of a path change as the number of hops on R which possibly change their next hop value. Flows through these routers may all have their paths changed. In Figure 3, L

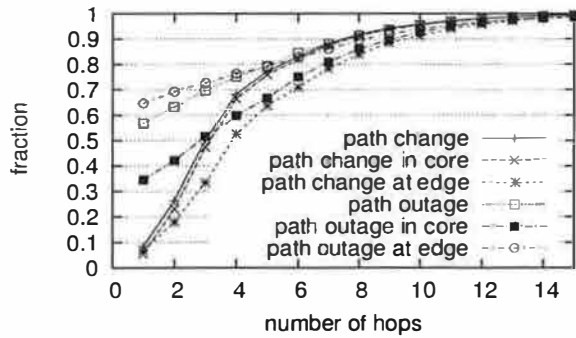


Figure 4: Scope of path changes and forward outages in number of hops

diverges from R at r_i and merges into R at r_l . All the hops before r_i or after r_l (including r_l) follow the same next hop towards the destination. So the hops which are possibly influenced by the path change and have different next hops are r_i , r_j and r_k .

In some cases, we may be able to use remote traceroutes to narrow the scope even further. For example, in Figure 3, if I , a traceroute from another ProbeD merges into R at r_k , a hop that is before r_l , we can eliminate r_k from the scope of the path change anomaly, since we know r_k has the same next hop value as it did in the reference path. We call I the *intercept path*. This method may still overestimate the scope of path change: it is possible, for example, that r_j 's next hop value is unaffected, but we cannot know this unless some traceroute merges into R at r_j .

Performing traceroute from multiple geographically diverse vantage points increases our chances of finding an intercept path. Our ability of narrowing the scope is affected by the location of the anomaly. If it is closer to the destination, we have a better chance of obtaining intercept paths by launching many forward traceroutes, and thus successfully reducing the scope of the anomaly. In contrast, if the anomaly is nearer our source, the chance of another traceroute merging into the reference path early is low.

Figure 4 shows the CDF of path change scope, measured in hop count. We can confine the scope of 68% of the path changes to within 4 hops. We do not count fluttering as path changes, since these would appear as a large number of path change anomalies, each with a very small scope. We also examine how many ASes the path change scope spans, shown in Table 8. Again, we can confine the scope of 57% of the path changes to within two ASes and 82% of them to within three ASes.

To gain some insight into the location of the anomalies, we also study whether the path change occurs near end hosts or in the middle of the path [7]. We measure the *distance* of a path change to the end host by averaging

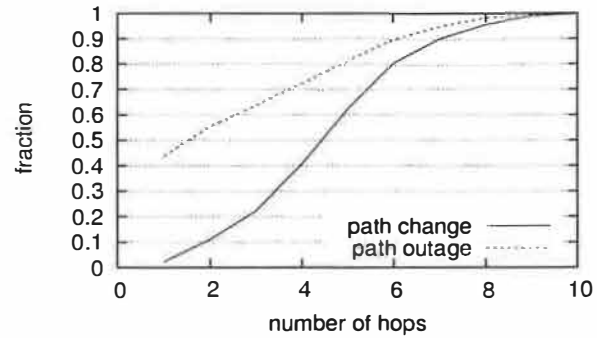


Figure 5: Distance of path changes and forward outages to the end hosts in number of hops

	Change	Fwd Outage
Total	120283	12740
No Ref Path	N/A	11181
1 AS	24418 (20%)	6534 (51%)
2 ASes	43909 (37%)	3413 (27%)
3 ASes	29426 (25%)	1321 (10%)
4 ASes	12603 (10%)	856 (7%)
5 ASes	6322 (5%)	411 (3%)
6 ASes	3605 (3%)	205 (2%)
Guessed Last Hop	N/A	1055
Scope Changed	4292 (4%)	1225 (10%)
Tier-1 AS	12374 (6%)	2746 (15%)
Tier-2 AS	43104 (23%)	3255 (18%)
Tier-3 AS	88959 (47%)	4638 (26%)
Tier-4 AS	8015 (4%)	3501 (19%)
Tier-5 AS	38313 (20%)	3838 (21%)
Total	190765	17978

Table 8: Summary of path change and forward outage. Some counts exceed 100% due to multiple classification.

ing the distances of all the routers within the path change scope. The *distance* of a router is defined as the minimum number of hops to either the source or the destination. Figure 5 plots the CDF of path change distances. As we can see, 60% of the path changes occur within 5 hops to the end hosts.

We can also use AS tiers to characterize network locations, so we map the routers within anomaly scopes to ASes and AS tiers. The breakdown of possibly affected routers by their AS tiers is shown in Table 8. The routers in Tier-3 are most likely to be affected by path changes, since they account for nearly half of the total. By comparison, the routers in tier-1 ASes are rather stable, though presumably they are traversed repeatedly by many paths.

In Figure 4, we see that path changes in the core net-

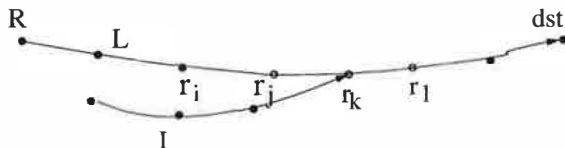


Figure 6: Confining the scope of forward outage

work have narrower scope than those in the edge. This is probably because the paths in the core network are likely to be traversed by traceroutes from many vantage points to reach the destination. In contrast, if a route change occurs near a local node, we have less chance of finding an intercept path that happens to merge into the reference path early. As a result, the anomaly scope in these cases is more loosely confined.

Since path change is a dynamic process and anomaly scopes may evolve over time, a measured anomaly scope should be viewed as a snapshot of which routers are affected when the traceroutes reach them. In Table 8, we show how many path changes have changed scope between the first and second sets of forward probes. We find that only 4% of them have changed during that period (mostly within one minute). In addition, 66% of the scope changes are due to local path changes instead of intercept path changes.

We now examine the effect of path changes on end-to-end performance. The effect of path changes on RTTs is relatively mild, as can be seen in Figures 8. The RTTs measured during path changes are only slightly worse than the RTTs measured in baseline traceroutes. But the loss rates during path changes can be very high. Nearly 45% of the path changes cause loss rates higher than 30%, which can significantly degrade TCP throughput.

7.2 Path Outage

We now focus on path outages and describe how to distinguish between forward and reverse path outages. In Figure 6, the local path L stops at r_i , which is an intermediate hop on the reference path R . At first glance, one might conclude that a failure has occurred after r_i on the forward path, which prevents the packets from going through; but other possibilities also exist—because Internet paths could be asymmetric, a failure on the reverse path may produce the same results. For example, if a shared link on the reverse paths from all the hops beyond r_i to the source has failed, none of the ICMP packets from those hops can return. Consequently, we will not see the hops after r_i .

If we have control of the destination, we can simply distinguish between forward and reverse path outages using ping [7]. However, since our clients are outside of PlanetLab and not under our control, we cannot perform pings in both directions, and must use other information to disambiguate forward path outages from reverse path

Route change	Unreachable	Fwd Timeout
12822 (54%)	2751 (11%)	8348 (35%)

Table 9: Breakdown of reasons for inferring forward outage

failures. Specifically, we can infer that the outage is on the forward path using the following rules:

- There is a route change on the forward path in addition to the outage.
- The local traceroute returns an ICMP destination unreachable message.
- The anomaly is reported as *timeouts on forward path*. As described in Section 3.4, MonD will report this type of anomaly when it infers ACK losses on the forward path from the local node to the client.

Table 9 shows the number of forward path outages inferred from each rule. As we can see, all three rules are useful in identifying forward outages. More than half of the outages are accompanied by route changes, as the failure information is propagated and some routers try to bypass the failure using other routes. Forward timeouts help us infer one third of the forward outages. This demonstrates the benefit of combining passive monitoring with active probing, since we would not have been able to disambiguate them otherwise.

7.2.1 Scope

To characterize the scope of path outages, we use a technique similar to the one we used to characterize the scope of path change. We define a path outage scope as the number of hops in a path that cannot forward packets to their next hop towards the destination. In Figure 6, R is the reference path and L is the local path. L stops at r_i , which is an intermediate hop of R . Hence, all the hops after r_i (including r_i) are possibly influenced by the outage and may not be able to forward packets to the next hops towards the destination. However, when we can find another intercept path, we can narrow the scope. For example, if I merges into R at r_k and reaches the destination, then only r_i and r_j can possibly be influenced by the outage. Again, this method might overestimate the scope of a path outage, for the same reasons described earlier on estimating a path change scope.

Note that unlike in previous work [6, 7], we use a set of routers to quantify the effect of path outages instead of just using the last hop where the traceroute stops. Since outage information is usually propagated to many routers, using only one router does not give us a sense of how many routers may have been affected by the outage.

In some cases, we may not have a complete baseline path which reaches the destination or the penultimate router. In these cases, we can not estimate the scope

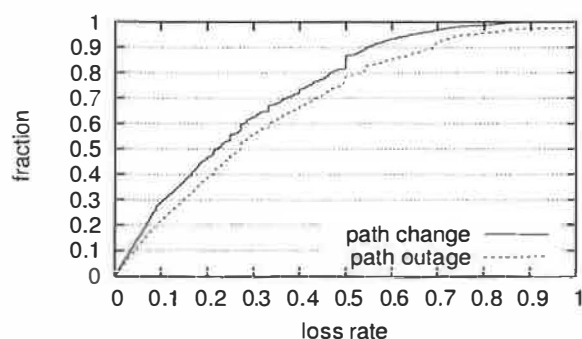


Figure 7: CDF of loss rates preceding path changes and forward outages

of the forward outage because we do not know the exact number of hops between the last hop of the baseline path and the destination. We only know that the anomaly occurs somewhere on the forward path. Among all the outages, about 47% have no complete reference path. In the following, we use only those with complete reference paths in the scope analysis.

In Figure 4, we plot the CDF of the number of hops in the forward outage scope. Compared with path change, we can confine the outage scope more tightly. Nearly 60% of the outages can be confined to within 1 hop and 75% of them can be confined to 4 hops.

We suspect that such tight confinement is due to last hop failures. In Figure 5, we plot the distances of forward outages to the end hosts. The distance of an outage is defined as the average distance of the routers within the outage scope to the end hosts, similar to the definition used for path change in Section 7.1. As we can see, 44% of the outages do occur at the last hop, allowing us to confine their scopes to 1 hop. This observation explains why the outages in the edge network are confined more tightly than those in core networks, as shown in Figure 4.

Excluding last hop failures, we can only confine 14% of the outages to one hop, a result that is slightly better than that for path changes. In general, the scopes of path outages tend to be smaller than those of path changes. Compared with path changes in Figure 5, path outages tend to occur much closer to the end hosts. More than 70% of the outages occur within 4 hops to the end hosts.

Table 8 gives the number of ASes that the outages span. Compared with path changes, we can confine a much higher percentage of outages (78%) within two ASes. If we examine the AS tiers where the affected routers are located, outages are spread out more evenly across tiers than path changes are. Paths in tier-1 ASes are the most stable and those in tier-3 ASes are most unstable. If we look at both Table 5 and Table 8, we note that paths in tier-3 ASes are most likely to be affected by all types of anomalies. They account for 40% of tempo-

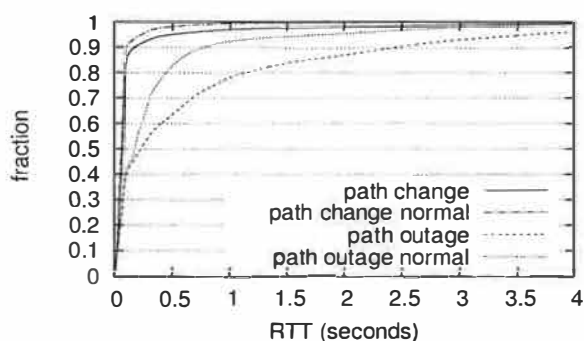


Figure 8: CDF of RTTs preceding path changes and forward outages vs. under normal conditions

rary loops, 46% of persistent loops, 47% of path changes and 26% of forward outages. In contrast, paths in tier-1 ASes are most stable.

Finally, in Table 8, we find that the scopes of 10% of the forward outages might have changed between the first and second set of forward probes, mostly due to local path changes. Another 8% of the forward outages have reference paths that do not terminate at the destinations. These last hops are considered firewalls based on the heuristic described in Section 6.

7.2.2 End-to-End Effect

We also study how path outages influence end-to-end performance. Not surprisingly, forward outages can be preceded by very high loss rates, which are slightly worse than those generated by path changes. The comparisons are shown in Figure 7. Similarly, outages tend to be preceded by much worse RTTs than path changes, as shown in Figure 8: 23% of the outages experience RTTs that are over one second, while only 7% do when there is no outage. The RTT variances can also be very high: 17% of them exceed 0.5 seconds.

8 Discussion

8.1 Bypassing Anomalies

In addition to characterizing anomalies, one of the goals of PlanetSeer is to provide benefits to the hosts running it. One possible approach is using the wide-area service nodes as an overlay, to bypass path failures. Existing systems, such as RON [1], bypass path failures by indirectly routing through intermediate nodes before reaching the destinations. Their published results show that around 50% of the failures on a 31-node testbed can be bypassed [7]. PlanetSeer differs in size and scope, since we are interested in serving thousands of clients that are not participants in the overlay, and we have a much higher AS-level coverage.

Determining how many failures can be bypassed in our model is more complicated, since we have no con-

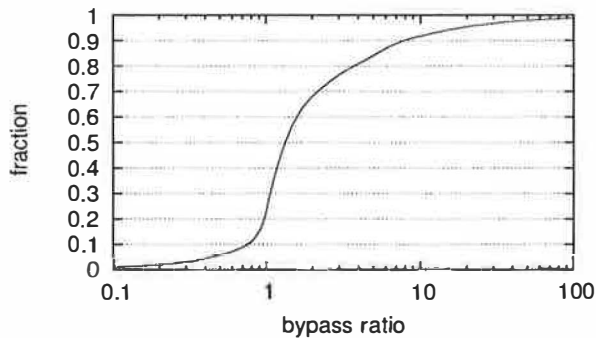


Figure 9: CDF of latency ratio of overlay paths to direct paths

trol over the clients. Clients that are behind firewalls and filter pings and traceroutes may be reachable from other overlay nodes, but we may not be able to confirm this scenario. As a result, we focus only on those destinations that are reachable in the baseline probes, since we can confirm their reachability during normal operation.

For this group of clients, we have a total of **62815** reachability failures, due to anomalies like path outages or loops. Of these failures, we find that some nodes in PlanetSeer are able to reach the destinations in **27263** cases, indicating that one-hop indirection is effective in finding a bypass path for **43%** of the failures.

In addition to improving the reachability of clients using overlay paths, the other issue is their relative performance during failures. We calculate a *bypass ratio* as the ratio between the minimum RTT of any of the bypass paths and the RTT of the baseline path. These results are shown in Figure 9, and we see that the results are moderately promising – 68% of the bypass paths suffer less than a factor of two in increased latency. In fact, 23% of the new paths actually see a latency *improvement*, suggesting that the overlay could be used for improving route performance in addition to failure resiliency. However, some paths see much worse latency degradation, with the worst 5% seeing more than a factor of 18 worse latency. While these paths may bypass the anomaly, the performance degradation will be very noticeable, perhaps to the point of unusability.

8.2 Reducing Measurement Overhead

While PlanetSeer’s combination of passive monitoring and distributed active probing is very effective at finding anomalies, particularly the short-lived ones, the probing traffic can be aggressive, and can come as a surprise to low-traffic sites that suddenly see a burst of traceroutes coming from around the world. Therefore, we are interested in reducing the measurement overhead while not losing the accuracy and flexibility of our approach. For example, we can use a single traceroute to confirm loops, and then decide if we want distributed traceroutes

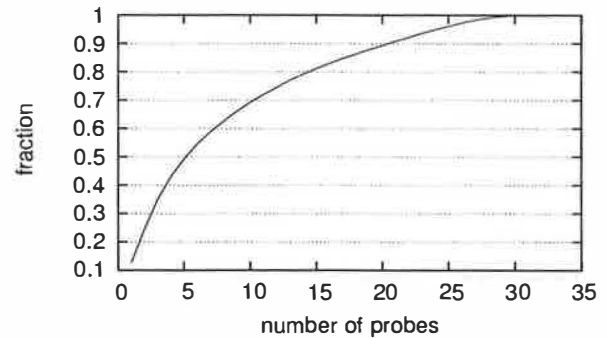


Figure 10: CDF of number of path examined before finding the intercept path

to test for the presence of correlated loops. Similarly, for path changes and outages, we can reduce the number of distributed traceroutes if we are willing to tolerate some inaccuracy in characterizing their scope. In Figure 10, we plot the CDF of the number of the probes from other vantage points we have to examine before we find the intercept traceroutes that can successfully narrow the scopes of the anomalies. Using only 15 vantage points, we achieve the same results as when using all 30 vantage points in 80% of the cases. We are interested in studying this issue further, so that we can determine which vantage points we need to achieve good results.

9 Related Work

There is extensive existing work on studying Internet path failure. Labovitz and Ahuja [15] studied inter-domain routing failures using BGP data collected from several ISPs and 5 network exchange points. They analyzed the temporal properties of failures, such as mean time to fail, mean time to repair and failure duration. They found that 40% of the failures are repaired in 10 minutes and 60% of them are resolved in 30 minutes. They also studied the intra-domain routing failures of a medium-sized regional ISP by examining the data from the trouble ticket tracking system managed by the Network Operation Center (NOC) of the ISP, together with the OSPF routing messages. Based on this data, they characterize the origins of failures into hardware, software and operational problems.

Iannaccone *et al.* investigated the failures in Sprint’s IP backbones using the IS-IS routing updates collected from three vantage points [12]. They examined the frequency and duration of failures inferred from routing updates and concluded that most failures are short-lived (within 10 minutes). They also studied the interval between failures. Again, their focus is on the temporal properties of failure.

Mahajan, Wetherall and Anderson [16] studied BGP misconfigurations using BGP updates from Route-

Views [18], which has 23 vantage points across different ISPs. They found BGP misconfigurations were relatively common and classified them into origin and export misconfigurations. Note that BGP misconfigurations may or may not be visible to end users. They observed that only 1 in 25 misconfigurations affect end-to-end connectivity.

More recently, Feldmann *et al.* have presented a methodology to locate the origin of routing instabilities along three dimensions in BGP: time, prefix, and view [9]. Their basic assumption is that an AS path change is caused by some instability either on the previous best path or the new best path. Caesar *et al.* [4] and Chang *et al.* [5] also propose similar approaches to analyze routing changes, although their algorithms are different in details.

All of the above failure studies are based on either inter-domain (BGP) or intra-domain (IS-IS, OSPF) routing messages, from which failures are inferred. Some of them require access to the ISP's internal data, such as trouble tickets. Our study complements this work by studying failures from an end-to-end perspective, and quantifies how failures affect end-to-end performance, such as loss rate and RTT.

There also has been much work on studying Internet failures through end-to-end measurement, and these have greatly influenced our approach. Paxson [19] studied the end-to-end routing behavior by running repeated traceroutes between 37 Internet hosts. His study shows that 49% of the Internet paths are asymmetric and visit at least one different city. 91% of the paths persist more than several hours. He used traceroutes to identify various routing pathologies, such as loops, fluttering, path changes and outages. However these traceroutes do not distinguish between forward and reverse failures.

Chandra *et al.* [6] studied the effect of network failures on end-to-end services using the traceroute datasets [19, 21]. They also used the HTTP traces collected from 11 proxies. They model the failures using their location and duration and evaluate different techniques for masking failures. However, the HTTP and traceroute datasets are independent. In comparison, we combine the passive monitoring data and active probing data, which allows us to detect failures in realtime and correlate the end-to-end effect with different types of failures. They also classify the failures into near-source, near-destination and in-middle by matching /24s IP prefixes with end host IPs. In contrast, we study the location of failures using both IP-to-AS mapping [17] and 5-tier AS hierarchies [23]. This allows us to quantify the failure locations more accurately and at a finer granularity.

Feamster *et al.* measured the Internet failures among 31 hosts using periodic pings combined with traceroutes [7]. They ping the path between a pair of nodes every 30 seconds, with consecutive ping losses trigger-

ing traceroutes. They consider the location of a failure to be the last reachable hop in traceroute and used the number of hops to closest end host to quantify the depth of the failure. They characterize failures as inter-AS and intra-AS and use one-way ping to distinguish between forward and reverse failures. They also examine the correlation between path failures with BGP updates.

Our work is partly motivated by these approaches, but we cannot use their methodology directly because of environmental differences. With the large number of clients that connect to our proxies, we can not afford to ping each of them frequently. Failure detection and confirmation are more challenging in our case, since many clients may not respond to pings (behind firewalls) or even are offline (such as dialup users). We infer anomalies by monitoring the status of active flows, which allows us to study anomalies on a much more diverse set of paths. We also combine the baseline traceroutes with passive monitoring to distinguish between forward and reverse failures and classify forward anomalies into several categories. Since where the failure appears may be different from where the failure occurs, we quantify the scope of failures by correlating the traceroutes from multiple vantage points, instead of using one hop (last reachable hop) as the failure location. Finally, we study how different types of anomalies affect end-to-end performance.

10 Conclusion

This paper introduces what we believe to be an important new type of diagnostic tool, one that passively monitors network communication watching for anomalies, and then engages widely-distributed probing machinery when suspicious events occur. Although much work can still be done to improve the tool—e.g., reducing the active probes required, possibly by integrating static topology information and BGP updates—the observations we have been able to make in a short time are dramatic.

- Passive monitoring allows us to detect more anomalies in less time: we have confirmed nearly 272,000 anomalies in three months. This is roughly 3,000 a day, and is 10 to 100 times more than reported previously. We also see a qualitative change, such as large numbers of ultrashort and temporary anomalies that last less than one minute.
- Due to our wide coverage, we see new failure distribution and location properties. Failures are heavily skewed, rather than pervasively distributed: Tier 3 seems to be the most problematic, accounting for almost half of the loops, path changes, and path outages that we see. Tier 1 ASes are generally the most stable.

- We provide some new measurements about routing loop behavior. Temporary loops have much longer lengths than persistent loops. 97% of the persistent loops consist of only 2 routers, but only 50% of temporary loops do. Many temporary loops span 4 routers. This makes sense since the more routers are involved in a loop, the less stable it is. Persistent loops are either resolved in a relative short time (54% last less than 30 minute) or continue for an extended period of time (23% last more than 7.5 hours). Our results confirm Paxson's findings that routing loops are correlated.
- Path changes exhibit different characteristics than outages. Outages appear closer to the edge of the network: 63% of outages occur within 3 hops to end hosts while the figure is 20% for path changes. Path changes tend to have wider impact: 57% of path changes can be confined to two ASes and 50% of them can be confined to within three hops, while the respective figures are 78% and 70% for outages. Path changes have a much milder effect on RTTs than outages while they both can incur high loss rates.
- Our measurements suggest less opportunity for indirection-based resiliency than previous studies: alternative routes are available only 43% of the time, and a significant fraction of them suffer from high latency inflation. These results stem from most outages occurring nearer the edge of the network than the core; redundancy is less available, and less practical when it is available.

We have shown that PlanetSeer provides an effective means to detect large numbers of anomalies with broad coverage, especially in the case of wide-area services that cannot rely on cooperation from one endpoint. In addition to the detection rate, the short delay between emergence and detection allows us to capture anomaly behavior more effectively, and our distributed framework provides improved characterization.

Acknowledgments

This research was supported in part by NSF grant CNS-0335214. We would like to thank KyoungSoo Park for his effort in keeping CoDeeN operational during this work. We thank our shepherd, Miguel Castro, for his guidance and helpful feedback, and we thank our anonymous reviewers for their valuable comments on improving this paper.

References

- [1] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient overlay networks. *ACM SOSP*, Oct. 2001.

- [2] S. Banerjee, T. G. Griffin, and M. Pias. The interdomain connectivity of PlanetLab nodes. In *Passive and Active Measurement Workshop*, April 2004.
- [3] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Mawrzoniak. Operating system support for planetary-scale network services. In *USENIX/ACM NSDI*, Mar. 2004.
- [4] M. Caesar, L. Subramanian, and R. H. Katz. Route cause analysis of Internet routing dynamics. In *Tech Report UCB/CSD-04-1302*, 2003.
- [5] D.-F. Chang, R. Govindan, and J. Heidemann. The temporal and topological characteristics of BGP path changes. In *IEEE ICNP*, Nov. 2003.
- [6] M. Dahlin, B. Chandra, L. Gao, and A. Nayate. End-to-end WAN service availability. *ACM/IEEE Trans. Netw.*, Apr 2003.
- [7] N. Feamster, D. G. Andersen, H. Balakrishnan, and M. F. Kaashoek. Measuring the effects of Internet path faults on reactive routing. In *ACM SIGMETRICS*, Jun 2003.
- [8] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True. Deriving traffic demands for operational IP networks: methodology and experience. *ACM SIGCOMM*, Aug. 2000.
- [9] A. Feldmann, O. Maennel, Z. Mao, A. Berger, and B. Maggs. Locating Internet routing instabilities. In *ACM SIGCOMM*, Aug 2004.
- [10] U. Hengartner, S. Moon, R. Mortier, and C. Diot. Detection and analysis of routing loops in packet traces. In *ACM IMW*, 2002.
- [11] IANA. Special-use IPv4 addresses. RFC 3330.
- [12] G. Iannaccone, C.-N. Chuah, R. Mortier, S. Bhattacharyya, and C. Diot. Analysis of link failures in an IP backbone. In *ACM IMW*, Nov 2002.
- [13] J. Postel. Internet control message protocol. RFC 792.
- [14] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian. Delayed Internet routing convergence. In *ACM SIGCOMM*, Sep 2000.
- [15] C. Labovitz, A. Ahuja, and F. Jahanian. Experimental study of Internet stability and wide-area backbone failures. Technical Report CSE-TR-382-98, University of Michigan, 1998.
- [16] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. In *ACM SIGCOMM*, 2002.
- [17] Z. Mao, J. Rexford, J. Wang, and R. H. Katz. Towards an accurate AS-level traceroute tool. In *ACM SIGCOMM*, 2003.
- [18] U. of Oregon RouteViews Project. <http://www.routeviews.org>.
- [19] V. Paxson. End-to-end routing behavior in the Internet. In *ACM SIGCOMM*, Aug 1996.
- [20] V. Paxson. End-to-end Internet packet dynamics. *IEEE/ACM Trans. Netw.*, 7(3), 1999.
- [21] S. Savage, A. Collins, and E. Hoffman. The end-to-end effects of Internet path selection. *ACM SIGCOMM*, Aug. 1999.
- [22] N. Spring, D. Wetherall, and T. Anderson. Scriptroute: A facility for distributed Internet measurement. *USITS*, March 2003.
- [23] L. Subramanian, S. Agarwal, J. Rexford, and R. H. Katz. Characterizing the Internet hierarchy from multiple vantage points. *IEEE INFOCOM*, June 2002.
- [24] Traceroute.Org. <http://www.traceroute.org>.
- [25] V. Paxson and M. Allman. Computing TCP's retransmission timer. RFC 2988.
- [26] L. Wang, V. Pai, and L. Peterson. The effectiveness of request redirection on CDN robustness. In *OSDI*, Dec. 2002.
- [27] Y. Zhang, N. Duffield, V. Paxson, and S. Shenkar. On the constancy of Internet path properties. *ACM IMW*, Nov. 2001.

Improving the Reliability of Internet Paths with One-hop Source Routing

Krishna P. Gummadi, Harsha V. Madhyastha,
Steven D. Gribble, Henry M. Levy, and David Wetherall
Department of Computer Science & Engineering
University of Washington
{gummadi, harsha, gribble, levy, djw}@cs.washington.edu

Abstract

Recent work has focused on increasing availability in the face of Internet path failures. To date, proposed solutions have relied on complex routing and path-monitoring schemes, trading scalability for availability among a relatively small set of hosts.

This paper proposes a simple, scalable approach to recover from Internet path failures. Our contributions are threefold. First, we conduct a broad measurement study of Internet path failures on a collection of 3,153 Internet destinations consisting of popular Web servers, broadband hosts, and randomly selected nodes. We monitored these destinations from 67 PlanetLab vantage points over a period of seven days, and found availabilities ranging from 99.6% for servers to 94.4% for broadband hosts. When failures do occur, many appear too close to the destination (e.g., last-hop and end-host failures) to be mitigated through alternative routing techniques of any kind. Second, we show that for the failures that *can* be addressed through routing, a simple, scalable technique, called *one-hop source routing*, can achieve close to the maximum benefit available with very low overhead. When a path failure occurs, our scheme attempts to recover from it by routing *indirectly* through a small set of *randomly chosen* intermediaries.

Third, we implemented and deployed a prototype one-hop source routing infrastructure on PlanetLab. Over a three day period, we repeatedly fetched documents from 982 popular Internet Web servers and used one-hop source routing to attempt to route around the failures we observed. Our results show that our prototype successfully recovered from 56% of network failures. However, we also found a large number of server failures that cannot be addressed through alternative routing.

Our research demonstrates that one-hop source routing is easy to implement, adds negligible overhead, and achieves close to the maximum benefit available to indirect routing schemes, *without* the need for path monitoring, history, or a-priori knowledge of any kind.

1 Introduction

Internet reliability demands continue to escalate as the Internet evolves to support applications such as banking and telephony. Yet studies over the past decade have consistently shown that the reliability of Internet paths falls far short of the “five 9s” (99.999%) of availability expected in the public-switched telephone network [11]. Small-scale studies performed in 1994 and 2000 found the chance of encountering a major routing pathology along a path to be 1.5% to 3.3% [17, 26].

Previous research has attempted to improve Internet reliability by various means, including server replication, multi-homing, or overlay networks. While effective, each of these techniques has limitations. For example, replication through clustering or content-delivery networks is expensive and commonly limited to high-end Web sites. Multi-homing (provisioning a site with multiple ISP links) protects against single-link failure, but it cannot avoid the long BGP fail-over times required to switch away from a bad path [12]. Overlay routing networks, such as RON, have been proposed to monitor path quality and select the best available path via the Internet or a series of RON nodes [2]. However, the required background monitoring is not scalable and therefore limits the approach to communication among a relatively small set of nodes.

This paper re-examines the potential of overlay routing techniques for improving end-to-end Internet path reliability. Our goal is to answer three questions:

1. What do the failure characteristics of wide-area Internet paths imply about the potential reliability benefits of overlay routing techniques?
2. Can this potential be realized with a simple, stateless, and scalable scheme?
3. What benefits would end-users see in practice for a real application, such as Web browsing, when this scheme is used?

To answer the first question, we performed a large-scale measurement study that uses 67 PlanetLab vantage

points to probe 3,153 Internet destinations for failures over seven days. Of these destinations, 378 were popular Web servers, 1,139 were broadband hosts, and 1,636 were randomly selected IP addresses. During the course of our 7-day study we observed more failures than the 31 node RON testbed saw in 9 months [7].

Our results show that end-to-end path availability varies substantially across different destination sets. On average, paths to popular Web servers had 99.6% availability, but paths to broadband hosts had only 94.4% availability. The vast majority of paths experienced at least one failure. Unfortunately, many failures are located so close to the destination that no alternative routing or overlay scheme can avoid them: 16% of failures on paths to servers and 60% of failures on paths to broadband hosts occur were last-hop or end-system failures. Effective remedies for these failures are increased end-system reliability and multi-homing.

To answer the second question, we use our measurement results to show that when an alternative path exists, that path can be exploited through extremely simple means. Inspired by the Detour study [21] and RON [2], we explore the use of a technique we call *one-hop source routing*. When a communication failure occurs, the source attempts to reach the destination *indirectly* through a small set of intermediary nodes. We show that a selection policy in which the source node chooses four potential intermediary nodes at random (called *random-4*) can obtain close to the maximum possible benefit. This policy gives well-connected clients and servers the ability to route around failures in the middle of the network *without* the need for complex schemes requiring *a priori* communication or path knowledge.

To answer the third question, we built and evaluated a prototype one-hop source routing implementation called SOSR (for Scalable One-hop Source Routing). SOSR uses the Linux netfilter/iptables facility to implement alternative packet routing for sources and NAT-style forwarding for intermediaries – both at user level. SOSR is straightforward to build and completely transparent to destinations. On a simple workload of Web-browsing requests to popular servers, SOSR with the random-4 policy recovered from 56% of network failures. However, many failures that we saw were application-level failures, such as server timeouts, which are not recoverable through any alternative routing or overlay schemes. Including such application-level failures, SOSR could recover from only 20% of the failures we encountered. The user-level perception of any alternative routing scheme is ultimately limited by the behavior of the servers as well as of the network.

The rest of the paper is organized as follows. We present our measurement study characterizing failures in the next section. Section 3 then shows the potential ef-

fectiveness of different practical policies for improving Internet path reliability. Section 4 describes the design, implementation, and evaluation of our prototype one-hop source routing system. We end with a discussion of related work (Section 5) and our conclusions (Section 6).

2 Characterizing Path Failures

This section describes our large-scale measurement study of Internet path failures. Our goals were: (1) to discover the frequency, location, and duration of path failures, and (2) to assess the potential benefits of one-hop source routing in recovering from those failures.

2.1 Trace methodology

From August 20, 2004 to August 27, 2004 we monitored the paths between a set of PlanetLab [18] *vantage points* and sets of destination hosts in the Internet. We periodically sent a probe on each path and listened for a response. If we received a response within a pre-defined time window, we declared the path to be normal. If not, we declared a *loss incident* on that path. Once a loss incident was declared, we began to probe the path more frequently to: (1) distinguish between a “true” path failure and a short-lived congestion episode and (2) measure the failure duration. We used traceroute to determine where along the path the failure occurred. In parallel, we also sent probes from the vantage point to the destination *indirectly* through a set of intermediary nodes. This allowed us to measure how often indirect paths succeeded when the default path failed.

2.1.1 Probes and traceroutes

Our probes consist of TCP ACK packets sent to a high-numbered port on the destination. Correspondingly, probe responses consist of TCP RST packets sent by the destination. We used TCP ACK packets instead of UDP or ICMP probes for two reasons. First, many routers and firewalls drop UDP or ICMP probes, or treat them with lower priority than TCP packets, which would interfere with our study. Second, we found that TCP ACK probes raise fewer security alarms than other probes. Before we included a candidate destination in our study, we validated that it would successfully respond to a burst of 10 TCP ACK probes. This ensured that destinations were not rate-limiting their responses, avoiding confusion between rate-limiting and true packet loss.

To determine *where* a failure occurred along a path, we used a customized version of traceroute to probe the path during a loss incident. Our version of traceroute uses TTL-limited TCP ACK packets, probing multiple hops along the route in parallel. This returns results much faster than the standard traceroute and permits us to determine the location of even short-lived failures.

2.1.2 Node selection

Initially, we selected 102 geographically distributed PlanetLab nodes as vantage points. Following the experiment, we examined the logs on each node to determine which had crashed or were rebooted during our trace. We then filtered the trace to remove any nodes with a total downtime of more than 24 hours, reducing the set to 67 stable vantage points for our analysis. We similarly selected 66 PlanetLab nodes to use as intermediaries, but only 39 of these survived crash/reboot post-filtering.

Using our vantage points, we monitored paths to three different sets of Internet hosts: popular Web servers, broadband hosts, and randomly selected IP addresses. A full list of IP addresses in each set and additional details describing our selection process are available at <http://www.cs.washington.edu/homes/gummadi/sosr>.

The members of each set were chosen as follows:

- We culled our popular server set from a list of the 2,000 most popular Web sites according to www ranking.com. Removing hosts that failed the TCP ACK rate-limit test and filtering duplicate IP addresses left us with 692 servers. The path behavior to a popular server is meant to be representative of the experience of a client when contacting a well-provisioned server.
- We selected our broadband hosts from an IP address list discovered through a 2002 crawl of Gnutella [20]. From that set, we removed hosts whose reverse DNS lookup did not match a list of major broadband providers (e.g., `adsl*bellsouth.net`) and again filtered those that failed the rate-limit test. Finally, we selected 2,000 nodes at random from those that survived this filtering. The path behavior to a broadband host is meant to be representative of a peer-to-peer application or voice-over-IP (VoIP).
- The random IP address set consists of 3,000 IP addresses that were randomly generated and that survived the rate-limit test. We use this set only as a basis for comparison.

We partitioned the destination sets across our initial vantage points such that each destination node was probed by only one vantage point. Because some of the vantage points were filtered from the trace due to failure or low availability, some of the destinations were consequently removed as well. Following this filtering, 378 servers, 1,139 broadband hosts, and 1,636 random IP addresses remained in the trace. Note that while we filtered vantage points and intermediaries for availability, we did *not* filter any of the destination sets beyond the initial

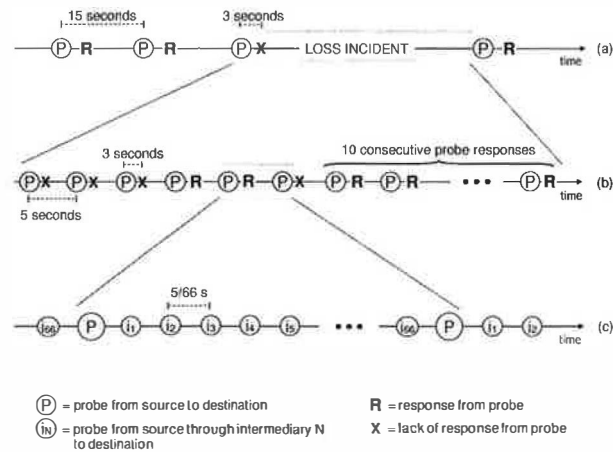


Figure 1: Probe timing. (a) The sequence of probes that are sent along each path during the trace. (b) A loss incident begins with a single probe loss, and ends after 10 consecutive successful probe responses. (c) For each of the first 10 probe intervals of a loss incident, we issued indirect probes through each of 66 intermediaries.

TCP ACK rate-limit test. As a consequence, some destinations crashed or otherwise shut down during the trace, causing last-hop path failures.

2.1.3 Probe timing

During the 7-day trace period, we probed each path every 15 seconds. If the vantage point failed to receive a response within 3 seconds, we declared a loss to have occurred. A single loss transitioned the path into a *loss incident*—an event initiated by a single probe loss and ended by the reception of ten consecutive probe responses (Figure 1a). While a path is in the midst of a loss incident, we probed every 5 seconds (Figure 1b). We also issued a traceroute from the vantage point to the destination at the start of the loss incident.

For each of the first 10 probe intervals during a loss incident, we also attempted to probe the destination *indirectly* through *each* of the 66 PlanetLab intermediaries selected at the beginning of the experiment. Thus, during one of these probe intervals, the vantage point emits a probe to an intermediary every 5/66th of a second (Figure 1c). We allow six seconds for a response to flow back from the destination through the intermediary to the vantage point; if no response is received in this time we declare a loss through that intermediary.

2.1.4 Failures vs. loss incidents

In principle, it may seem simple to declare a path failure when some component of a path has malfunctioned and all packets sent on that path are lost. In practice, however, failures are more complex and difficult to define. For example, packet loss may be due to a true long-term failure or a short-term congestion event. In general,

characteristic	servers	broadband	random
paths probed	378	1,139	1,636
vantage points	67	67	67
failure events	1,486	7,560	10,619
failed paths	294	999	1,395
failed links	337	1,052	1,455
classifiable failure events	962	5,723	7,024
last-hop	151 (16%)	3,406 (60%)	2,568 (37%)
non-last-hop	811 (84%)	2,317 (40%)	4,456 (63%)
unclassifiable failure events	524	1,837	3,595

Table 1: **High-level characterization of path failures, observed from 08/20/04 to 08/27/04.** We obtained path information using traceroutes. A failure is identified as a last-hop failure when it is attributable to either the access link connecting the destination to the network or the destination host itself.

any operational definition of failure based on packet loss patterns is arbitrary.

We strove to define failure as a sequence of packet losses that would have a significant or noticeable application impact. We did not want to classify short sequences of packet drops as failures, since standard reliability mechanisms (such as TCP retransmission) can successfully hide these. Accordingly, we elevated a loss incident to a failure if and only if the loss incident began with three consecutive probe losses *and* the initial traceroute failed. We defined the failure to last from the send of the first failed probe until the send of the first of the ten successful probes that terminated the loss incident. For example, the loss incident shown in Figure 1b corresponds to a failure that lasted for 30 seconds.

2.2 Failure characteristics

Table 1 summarizes the high-level characteristics of the failures we observed, broken down by our three destination sets. In the table we show as *classifiable* those failures for which our modified traceroute was able to determine the location of the failure; the remainder we show as *unclassifiable*. The classifiable failures are further broken down into *last-hop failures*, which are failures of either the end-system or last-hop access link (we cannot distinguish the two), and *non-last-hop failures*, which occurred within the network.

For the popular servers, we saw 1,486 failures spread over 294 paths and 337 links along these paths. Of the 962 classifiable failures, 811 (84%) occurred within the network, while 16% were last-hop failures. On average, a path experienced 3.9 failures during the week-long trace, of which 0.4 were last-hop failures, 2.1 were non-last-hop failures, and 1.4 were unclassifiable.

For the broadband hosts we saw 7,560 failures of which 5,723 were classifiable. On average, a broadband path experienced 6.6 failures over the week, nearly dou-

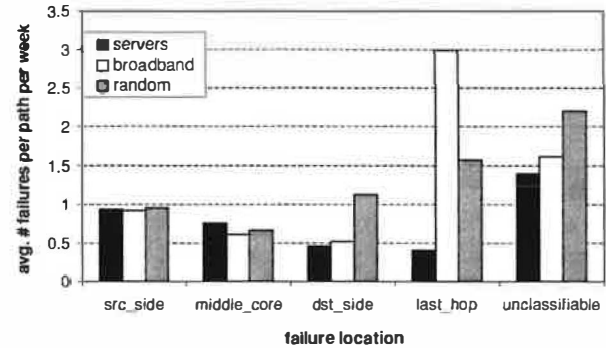


Figure 2: **Location of failures.** Failures are spread throughout the Internet for all three destination sets. Last-hop failures dominate other failures for broadband hosts, but for popular servers, last-hop failures are rare.

ble that of the popular server set. Of these 6.6 failures, 3.0 were last-hop, 2.0 non-last-hop, and 1.6 were unclassifiable. Comparing server and broadband paths, we saw approximately the same rate of non-last-hop failures, but broadband paths showed a much higher rate of last-hop failures (0.4 per path per week for servers, and 3.0 per path per week for broadband). Therefore, the network behaved similarly for broadband and popular server hosts, except over the last-hop.

2.2.1 Location of failures

To describe the failure locations in a meaningful way, we divide each path into four parts: *last_hop*, *middle_core*, *src_side*, and *dst_side*. Last-hop are either end-system failures or last-hop access-link failures. Middle-core failures occur in the “core of the Internet,” which we define as the *Tier1* domains. These are the few important domains, such as AT&T and Sprint, through which the vast majority of all Internet paths pass. We identify them using the methodology of Subramanian et al. [23]. Src_side and dst_side are therefore the remaining path segments between the core and source, or core and destination, respectively. If traceroute could not classify the failure location, we labeled it “unclassifiable.”

Figure 2 shows the distribution of failures across these categories. Failures are spread throughout the Internet, and all three data sets observe approximately equal source-side and core failures. For popular servers, there are relatively few last-hop failures, and in fact the last-hop appears to be more reliable than the rest of the Internet! This is strong evidence that techniques such as one-hop source routing can improve end-to-end availability for server paths, as it targets these non-last-hop failures. For broadband hosts, however, last-hop failures dominate all other failures, and accordingly we should expect less of a benefit from one-hop source routing.

Not surprisingly, the random IP destination set behaves in a manner that is consistent with a blend of

	servers	broadband	random
average path downtime	2,561 secs	33,630 secs	10,904 secs
median path downtime	333 secs	981 secs	518 secs
average failure duration	651 secs	5,066 secs	1,680 secs
last-hop	3,539 secs	8,997 secs	3,228 secs
non-last-hop	339 secs	859 secs	735 secs
unclassifiable	302 secs	3,085 secs	1,744 secs
median failure duration	73 secs	75 secs	72 secs
last-hop	113 secs	100 secs	61 secs
non-last-hop	70 secs	64 secs	66 secs
unclassifiable	70 secs	67 secs	85 secs

Table 2: **Path downtime and failure durations.** This table shows average and median path downtime, as well as average and median failure durations, for our three destination sets. The downtime for a path is the sum of all its failure durations.

server-like and broadband-like hosts. Somewhat surprisingly, however, the random IP set sees a greater rate of destination-side failures than both servers and broadband hosts. We do not yet have an explanation for this.

2.2.2 Duration of failures

In Table 2, we show high-level statistics that characterize failure duration and path availability in our trace. The average path to a server is down for 2,561 seconds during our week long trace, which translates into an average availability of 99.6%. In comparison, the average path to a broadband host is down for 33,630 seconds during trace, leading to an average availability of 94.4%. Paths to broadband hosts are an order of magnitude less available than paths to server hosts. This is unsurprising, of course, since broadband hosts are less well maintained, tend to be powered off, and likely have worse quality last-hop network connections.

The median path availability is significantly better than the average path availability, suggesting that the distribution of path availabilities is non-uniform. Figure 3 confirms this: for all three destination sets, more than half of the paths experienced less than 15 minutes of downtime over the week. As well as being generally less available than server paths, a larger fraction broadband paths suffer from high unavailability: more than 30% of broadband paths are down for more than an hour, and 13% are down for more than a day (not shown in graph).

Table 2 also shows the average and median failure durations. On paths to servers, the average failure lasted for just under 11 minutes; in comparison, on paths to broadband hosts, the average failure lasted for 84 minutes. For both destination sets, last-hop failures lasted approximately an order of magnitude longer than non-last-hop failures. Unfortunately, this reduces the potential effectiveness of one-hop source routing. Last-hop failures can last for a long time, and they are also hard to

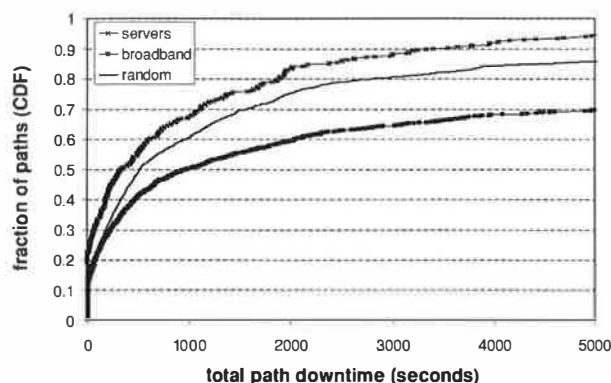


Figure 3: **Availability of paths (CDF).** The cumulative distribution of total downtime experienced by the paths during our trace, for each of our three destination sets.

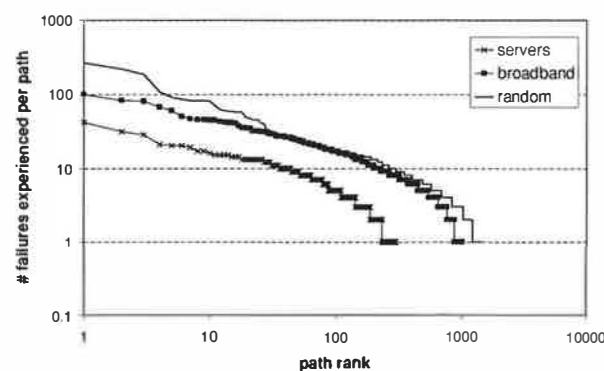


Figure 4: **Frequency of path failures.** Paths are ordered by the number of failures experienced. Most paths experience few failures, but a small number of paths experience many failures. Note that both axes are in log-scale.

route around. Like path availability, failure duration has a non-uniform distribution – median failure durations are significantly lower than average failure durations.

2.2.3 Frequency of failures

We can compute the number of failure-free paths by subtracting the number of failed paths from the number of paths probed (both shown in Table 1). This reveals that only 22% of server paths, 12% of broadband paths, and 15% of random paths were failure-free: most paths in each destination set experienced at least one failure.

Figure 4 plots the number of failures each path experienced, using a log-scale on both axes. Each destination set is sorted in rank order from most-failing to least-failing path. This graph demonstrates two points: (1) a small number paths experience a very large number of failures, and (2) most paths experienced a small but non-zero number of failures. Additional analysis (not shown) also demonstrates that for broadband hosts, the failure-prone paths tend to fail on the last-hop, while for servers, the failure-prone paths tend to fail uniformly across the

recoverable failures	servers	broadband	random
src_side	54% (189 of 353)	55% (577 of 1042)	54% (838 of 1548)
middle_code	92% (262 of 284)	90% (616 of 686)	94% (1017 of 1078)
dst_side	79% (138 of 174)	66% (391 of 589)	65% (1202 of 1830)
last_hop	41% (62 of 151)	12% (406 of 3406)	24% (606 of 2568)
unclassifiable	63% (332 of 524)	54% (983 of 1837)	58% (2096 of 3595)
all	66% (983 of 1486)	39% (2973 of 7560)	54% (5759 of 10619)

Table 3: **Potential effectiveness of one-hop source routing.** Source routing can help recover from 66% of all failures on paths to servers, but fewer on paths to broadband hosts. Last-hop failures tend to confound recovery, while core failures are more recoverable.

Internet, favoring neither the source-side, nor the core, nor the destination-side.

2.3 The potential of one-hop source routing

As previously described, during a loss incident we probed the destination indirectly through each of 39 intermediaries. If any of these indirect probes were successful, we considered the path to be recoverable using one-hop source routing. If not, we considered it to be unrecoverable. Note that this definition of recoverable provides an upper bound, since in practice an implementation is not likely to try such a large number of intermediaries when attempting to route around a failure.

The results of this experiment, shown in Table 3, indicate that 66% of all failures to servers are potentially recoverable through at least one intermediary. A smaller fraction (39%) of broadband failures are potentially recoverable. For all destination sets, one-hop source routing is very effective for failures in the Internet core, but it is less effective for source-side or destination-side failures. Somewhat surprisingly, some last-hop failures are recoverable. In part, this is due to multi-homing: i.e., there may be a last-hop failure on the default path to a destination, but a *different* last-hop link may be accessible on a different path through a destination. However, this is also due in part to our failure definition. If a last-hop link is not dead but merely “sputtering,” sometimes probes along the default path will fail while an intermediary will be more “lucky” and succeed.

2.4 Summary

Our study examined failures of Internet paths from 67 vantage points to over 3,000 widely dispersed Internet destinations, including popular servers, broadband hosts, and randomly selected IP addresses. Overall, we found that most Internet paths worked well: most paths only

experienced a handful of failures, and most paths experienced less than 15 minutes of downtime over our week-long trace. But failures do occur, and when they do, they were widely distributed across paths and portions of the network. However, broadband hosts tend to experience significantly more last-hop failures than servers, and last-hop failures tend to last long.

These failure characteristics have mixed implications for the potential effectiveness of one-hop source routing. Since server path failures are rarely on the last hop, there should be plenty of opportunity to route around them. Indeed, our initial results suggest that one-hop source routing should be able to recover from 66% of server path failures. In contrast, since broadband path failures are often on the last hop, there is less opportunity for alternative routing. Our results show that one-hop source routing will work less than 39% of the time in this case.

In the next section of the paper, we will examine one-hop source routing in greater detail, focusing initially on its potential for improving server path availability. Our goal in the next section is to use our trace to hone in on an effective, but practical, one-hop source routing policy. By effective, we mean that it successfully routes around recoverable failures. By practical, we mean that it succeeds quickly and with little overhead.

3 One-Hop Source Routing

We have seen that 66% of all popular server path failures and 39% of all broadband host path failures are potentially recoverable through at least one of the 39 pre-selected intermediary nodes. This section investigates an obvious implication of this observation, namely, that *one-hop source routing* is a potentially viable technique for recovering from Internet path failures.

One-hop source routing is conceptually straightforward, as shown in Figure 5. After a node detects a path failure, it selects one or more intermediaries and attempts to reroute its packets through them. If the resulting indirect path is sufficiently disjoint from the default route, the packets will flow around the faulty components and successfully arrive at the destination. Assuming that the reverse path through the intermediary also avoids the fault, end-to-end communication is restored.

This approach raises several questions. Given a set of potential intermediaries for a failed path, how many of them on average will succeed at contacting the destination? What policy should the source node use to select among the set of potential intermediaries? To what extent does the effectiveness of one-hop source routing depend on the location of the failure along the path? Does *a priori* knowledge of Internet topology or the ability to maintain state about previous failures increase the effectiveness of a policy? When should recovery be initiated

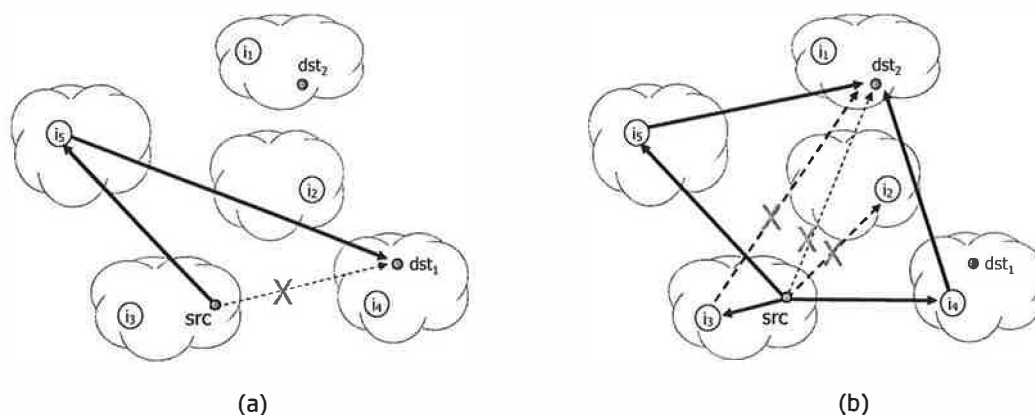


Figure 5: **One-hop source routing.** (a) The source (*src*) experiences a path failure to destination *dst*₁, but it successfully routes through intermediary *i*₅. (b) The source experiences a path failure to destination *dst*₂. It uses a more aggressive recovery policy by simultaneously routing to intermediaries *i*₂, *i*₃, *i*₄, and *i*₅. The path to intermediary *i*₂ experiences a path failure of its own, as does the path from intermediary *i*₃ to the destination. Fortunately, the source is able reach *dst*₂ through both *i*₄ and *i*₅.

and when should recovery attempts be abandoned? The remainder of this section answers these questions.

3.1 Methodology

To answer these questions we rely on the data described in Section 2. As previously noted, following each failure we sent probe messages from the source to 39 PlanetLab intermediaries. The intermediaries then probed the destination and returned the results. If the source heard back from an intermediary *before* it heard back directly from the (recovered) destination, then we considered that intermediary to be successful. Thus, for each default-path failure, we were able to determine *how many* of the 39 PlanetLab intermediaries *could have been used* to route around it.

From this data we can analyze the effectiveness of policies that route through specific subsets of the intermediaries. For example, one policy might route through a single, randomly chosen intermediary; another policy might route through two preselected intermediaries in parallel, and so on. We can therefore compare various policies by simulating their effect using the data from our intermediate-node measurements.

3.2 What fraction of intermediaries help?

How many of the intermediaries succeed in routing around a particular failure depends on a number of factors, including the positions of the source, the destination, and the intermediaries in the network. For example, some intermediaries may not divert the packet flow sufficiently, either failing to pull packets from the default path before the fault or failing to return them to the default path after the fault. This can be seen in Figure 6a, where the route from *src* to *dst* fails due to the failure

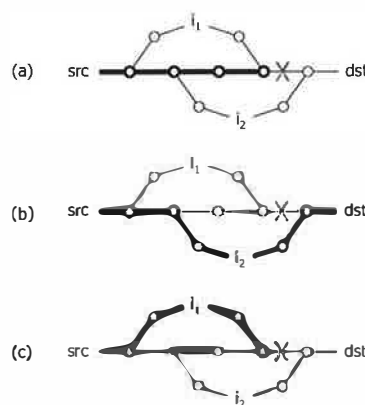


Figure 6: **Disjoint paths.** (a) The default path to the destination fails due to a faulty link. (b) Routing through intermediary *i*₂ would succeed, since the diverted path is disjoint from the faulty link. (c) Routing through intermediary *i*₁ would fail, since the diverted path rejoins the default path before the faulty link.

marked “X.” An attempt to re-route through intermediary *i*₂ would succeed (Figure 6b). However, routing through *i*₁ would fail (Figure 6c), because *i*₁’s path to *dst* joins *src*’s path to *dst* *before* the failure.

As described above, for each detected failure we counted the number of “useful intermediaries” through which the source node could recover. Note that we continue attempting to recover until either an intermediary succeeds or the default path self-repairs, up to a maximum of 10 probe intervals. If the default path self-repairs before any intermediary succeeds, we do not classify the event as a recovered failure.

Figure 7(a) shows the results for popular servers, grouped by the number of useful intermediaries on the

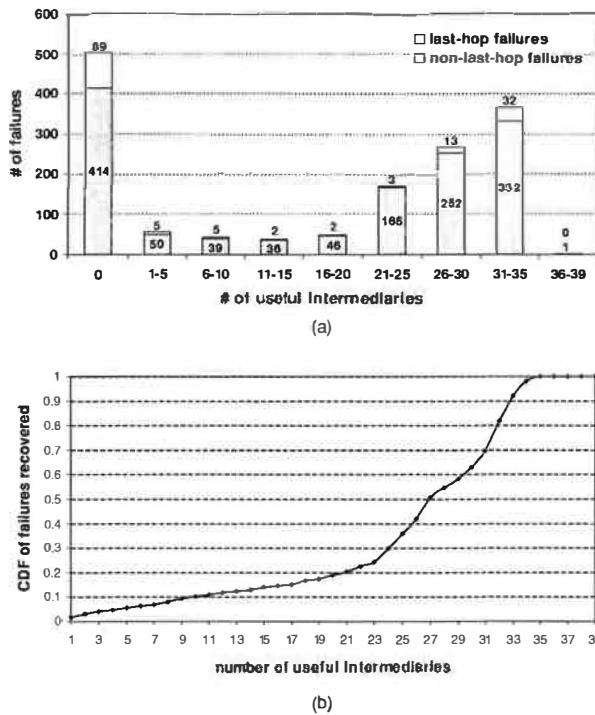


Figure 7: **The number of useful intermediaries.** For each failure, we measured the number of useful intermediaries in our candidate set of 39. (a) This histogram shows the aggregated results for popular servers only. For example, we observed 168 failures (165 non-last-hop and 3 last-hop) for which there were exactly 21-25 useful intermediaries. (b) A CDF of the same data for the recoverable failures only.

x-axis. Out of the 1486 failures, 503 (34%) could not recover through any intermediary, as shown by the left-most bar. Last-hop failures accounted for 89 of those unrecoverable failures.

Figure 7(b) presents a CDF of this data for the remaining 983 failures (66%) that *were* recoverable. The figure shows that 798 (81%) of these failures could be recovered through *at least* 21 of the 39 intermediaries. It's clear, then, that a significant fraction of failures are recoverable through a large number of intermediaries. However, there are also failures for which only a small number of intermediaries are useful. For example, 55 (5.6%) of the 983 recoverable failures could be recovered through only 1-5 nodes. Thus, some recoverable failures require a careful choice of intermediary.

None of the failures were recoverable through more than 36 of the 39 intermediaries. Investigating further, we found that four PlanetLab intermediaries were subject to a routing policy that prevented them from communicating with the vast majority of our destinations. If we exclude these nodes from consideration, many failures would be recoverable through all 35 of the remaining intermediaries. However, in many cases, there were still a

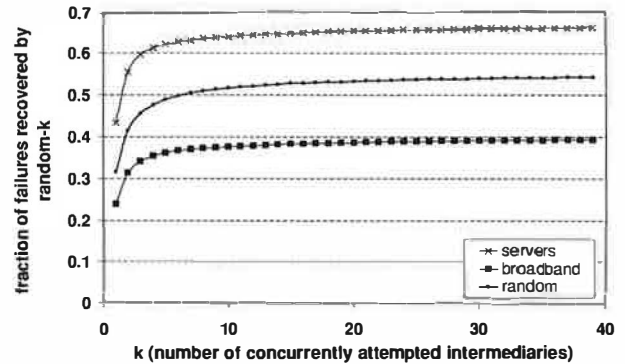


Figure 8: **The effectiveness of random-k.** This graph shows the effectiveness of random-k at recovering from failures as a function of k (the number of randomly selected intermediaries the source tries concurrently).

few intermediaries that should be avoided.

3.3 Is random-k an effective policy?

The results in Figure 7 suggest that a very simple strategy for selecting intermediaries may work well. Similar in spirit to randomized load-balancing [15, 6], a source should be able to avoid failures by randomly picking k intermediaries through which to attempt recovery. The source could send packets through all k intermediaries in parallel and then route through the intermediary whose response packet is first returned.

To evaluate this strategy, we examine a policy in which the random selection is done once for each failure instance. When a failure is detected, the source selects a set of k random intermediaries. During the failure, if none of the k intermediaries succeed on the first attempt, the source continues to retry those same intermediaries. At the next failure, the source selects a new set of k random intermediaries. We call this policy *random-k*.

Since many intermediaries can be used to avoid most recoverable faults, even a single random selection (random-1) should frequently succeed. By selecting more than one random intermediary, the source ensures that a single unlucky selection is not fatal. However, as there are some failures for which only a few specific intermediaries are helpful, picking a small number of random intermediaries will not always work.

Figure 8 shows the effectiveness of random-k as a function of k . For popular servers, random-1 can route around 43% of all failures we observed. By definition, random-39 can route around all recoverable failures (66% of all failures for popular servers). The “knee in the curve” is approximately $k = 4$: random-4 can route around 61% of all failures (92% of all recoverable failures) for popular servers. From this, we conclude that random-4 makes a reasonable tradeoff between effort (the number of concurrent intermediaries invoked per re-

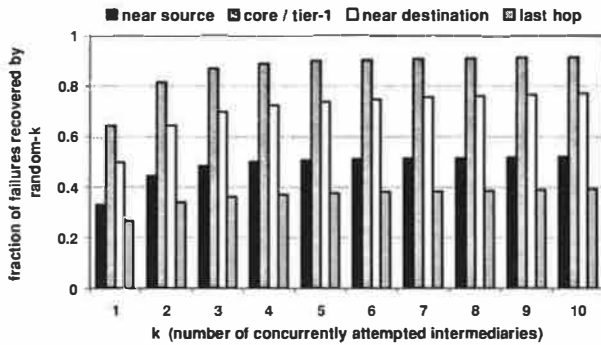


Figure 9: **Failures random-k can handle.** This graph breaks Figure 8 data down according to the classes of failures from which random-k recovers, for servers. Random-k can recover from most failures in the core, and near the destination, but it is less capable at handle failures near the source or on the last hop.

covery attempt) and the probability of success.

3.4 Which failures can random-k handle?

As we showed in Section 2, the location of a failure has a significant impact on the likelihood that one-hop source routing can recover. For example, last-hop failures are much harder to recover from than core failures. To understand the impact of failure location on random-k’s ability to recover, we classified recovery attempts according to the failure location. Figure 9 shows the same data as Figure 8 broken down according to this classification, but for popular servers only.

Random-k recovers poorly from near-source and last-hop failures, as shown in the figure. For example, random-4 recovers from only 37% of last-hop failures and 50% of near-source. However, random-4 is *very successful* at coping with the other failure locations, recovering from 89% of middle_core and 72% of near-destination failures. Intuitively, the Internet core has significant path diversity, therefore a failure in the core is likely to leave alternative paths between many intermediaries and the source and destination. However, the closer the failure is to the source or the destination, the more intermediaries it will render ineffective.

3.5 Are there better policies than random-k?

Figure 8 shows that random-k is a very effective policy: there is little room to improve above random-k before “hitting the ceiling” of recoverable failures. But can we be smarter? This subsection explores two alternative policies that use additional information or state to further improve on random-k. These policies might not be practical to implement, as they require significant amounts of prior knowledge of Internet topology or state. Nonetheless, an analysis of these policies offers additional insight

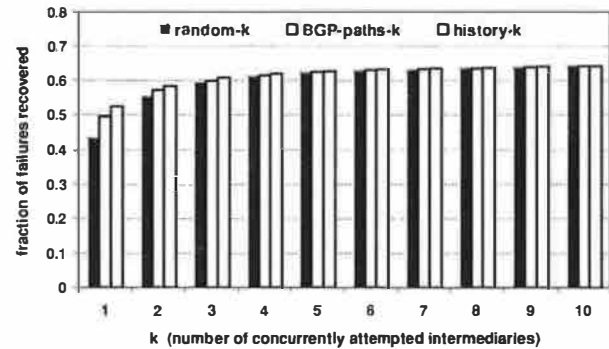


Figure 10: **Effectiveness of alternative policies.** This graph shows the effectiveness of our alternative policies as a function of k . We include random-k for comparison.

into one-hop source routing. Like random-k, each of the alternative policies selects k intermediaries through which to route concurrently for each recovery attempt. The two additional policies we consider are:

1. **History-k.** In this policy, we assume that the source node remembers the intermediary that it most recently used to recover from a path failure for each destination. When the source experiences a path failure, it selects $k - 1$ intermediaries at random, but it chooses this recently successful intermediary as the k^{th} intermediary. If the source has never experienced a failure for the destination, this policy reverts to random-k. The rationale for this policy is that an intermediary that previously provided a sufficiently disjoint path to a destination is likely to do so again in the future.
2. **BGP-paths-k.** In this policy, we assume that the source is able to discover the path of autonomous systems (ASs) between it and the destination, and between all intermediaries and the destination, as seen by BGP. For each intermediary, the source calculates how many ASs its path has in common with the intermediary’s path. When the source experiences a path failure to the destination, it orders the intermediaries by their number of common ASs and selects the k intermediaries with the smallest number in common. The rationale for this policy is that the source wants to use the intermediary with the “most disjoint” path to the destination in an attempt to avoid the failure.

Figure 10 shows how effective these policies were in recovering from failures. While there is some measurable difference between the policies for low values of k , this difference diminishes quickly as k increases. There is some benefit to using a more clever policy to pick the “right” intermediary; however, slightly increasing the number of intermediaries is more effective than using a

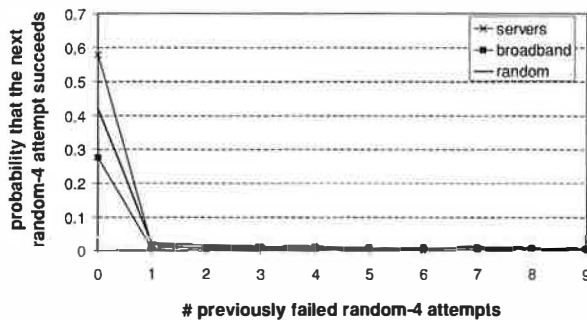


Figure 11: **The diminishing returns of repeated recovery attempts.** The probability that the next random-4 attempt will succeed, as a function of the number of previously failed attempts. After a single attempt, the probability of future success plummets.

smarter policy. For example, though BGP-paths-1 beats random-1, it does not beat random-4. Furthermore, unlike BGP-paths, random-4 requires no prior knowledge and no overhead for acquiring that knowledge.

These more sophisticated policies do have some positive effect on the ability to recover from path failures. They essentially attempt to bias their intermediary selection towards “better” intermediaries, and this biasing has some value. However, the goal of path failure recoverability is to avoid bad choices, rather than finding the best choice. Even with small k , random- k is extremely unlikely to select only bad choices, which is why it is competitive with these other strategies.

3.6 How persistent should random-4 be?

So far, we allow a source node recovering from a path failure to continue issuing random-4 attempts every 5 seconds until: (1) one of the four randomly selected intermediaries succeeds, (2) the path self-repairs, or (3) ten attempts have failed. We now consider the question of whether the source node should give up earlier, after fewer attempts. To answer this, we calculated the probability that the next random-4 attempt would succeed but the default path remains down, as a function of the number of previously failed random-4 attempts.

Figure 11 shows the results. Immediately after noticing the failure, random-4 for popular servers has a 58% chance of recovering before the path self-repairs; for broadband hosts, this number is 28%. However, after a single failed random-4 attempt, the chance that the next attempt succeeds before the path self-repairs plummets to 1.6% for servers and 0.8% for broadband hosts. There is little reason to try many successive random-4 attempts; the vast majority of the time, the default path will heal before a future random-4 attempt succeeds.

In a sense, random-4 is an excellent failure detector: if a random-4 attempt fails, it is extremely likely that the destination truly is unreachable. To be conservative,

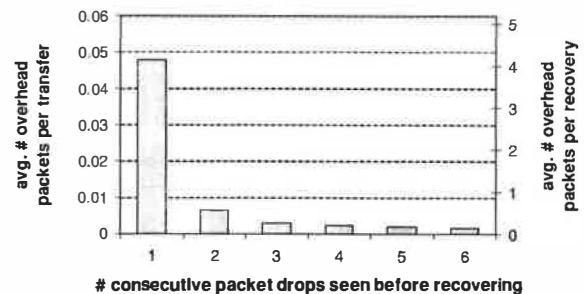


Figure 12: **The cost of being eager.** Average number of overhead packets per connection and per recovery invocation, as a function of the number of consecutive packet drops seen before random-4 recovery is invoked. Because failures happen rarely, recovery is inexpensive.

though, we allow four random-4 attempts before giving up on intermediary-based recovery.

3.7 How eager should random-4 be?

Our methodology defines a failure to have occurred after three consecutive packet drops and a failed traceroute. There is nothing preventing a source from invoking random-4 before making this determination, however. For example, a source may choose to invoke random-4 after seeing only a single packet drop. However, this approach might potentially confuse short-term congestion for failure and needlessly introduce recovery overhead.

To investigate this apparent tradeoff between recovery time and overhead, we used our measurement data to calculate the overhead of recovery relative to hypothetical background HTTP traffic. To do this, we consider each 15-second probe attempt to be a hypothetical HTTP request. We estimate that the transfer of a 10KB Web page, including TCP establishment and teardown, would require 23 IP packets. Using this estimate, we can calculate the amount of packet overhead due to random-4 recovery attempts.

Figure 12 shows this overhead as a function of how eagerly recovery is invoked. The left-hand y-axis shows the average number of additional packets sent per HTTP transfer, and the right-hand y-axis shows the average number of additional packets sent per transfer for which recovery was invoked. As our previous results suggested, we abandon after four repeated random-4 attempts.

Failures occur rarely and there is no overhead to pay when the default path succeeds. Additionally, when failures do occur, random-4 usually succeeds after a single attempt. For these two reasons, there is very little overhead associated with the average connection: only 0.048 additional packets per connection on average (on top of the estimated 23 IP packets), assuming recovery begins immediately after a lost packet. Even when failures occur, there are only 4.1 additional packets on average.

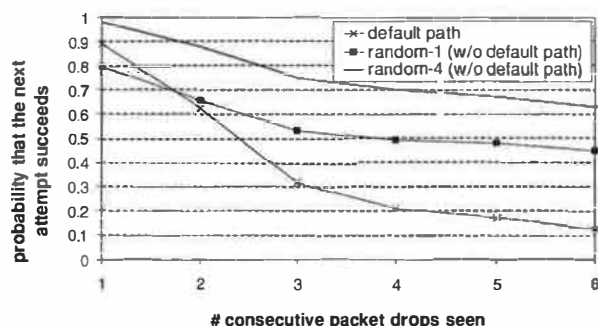


Figure 13: **The benefit of being eager.** This graph compares the likelihood that the default-path, a single-intermediary, and four intermediaries will succeed as a function of the number of consecutive packet drops observed. After just two observed packet drops, random-4 has a significantly higher probability of success than the default path.

There is therefore little cost to being eager.

Figure 13 demonstrates why it is useful to recover eagerly. We compare the probability that on the next attempt (1) the default-path will succeed, (2) a single randomly chosen intermediary will succeed, and (3) four randomly chosen intermediaries will succeed, as a function of the number of previously observed consecutive packet drops along the default path. After a single packet drop, all three strategies have an equally high probability of succeeding on the next attempt. But, with additional packet drops, the probability that the default path succeeds quickly decays, while random-1 and random-4 are more likely to work.

Note that random-1 has a slightly lower probability of succeeding than the default path after only one packet drop. For random-1, both the source to intermediary and the intermediary to destination paths must work, while for default-path, only the source to destination path must work.

There is a benefit to recovering early, and as we previously showed, there is very little cost. Accordingly, we believe random-4 should be invoked after having observed just a single packet drop.

3.8 Putting it all together

Our results suggest that an effective one-hop source routing policy is to begin recovery after a single packet loss, to attempt to route through both the default path and four randomly selected intermediaries, and to abandon recovery after four attempts to each of the randomly selected intermediaries fail, waiting instead for the default path to recover itself. For the remainder of this paper, we will refer to this set of policy decisions as “random-4.”

We now ask what the user experience will be when using random-4. To answer this, we measured how long the user must wait after experiencing a path failure for

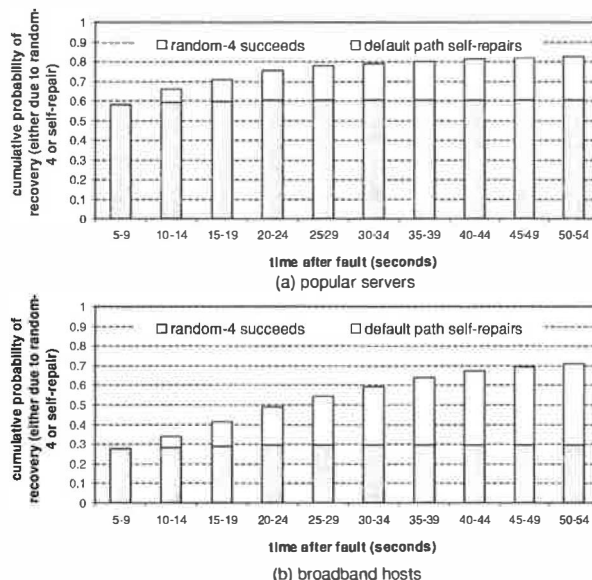


Figure 14: **Recovery latency.** This graph plots the cumulative probability that a failed path has recovered – due either to random-4 succeeding or self-repair of the default path – as a function of time, for (a) popular servers, (b) broadband hosts.

end-to-end connectivity to be re-established, due either to the success of random-4 or path self-repair.

Figure 14 shows the results. For popular servers, we see that 58.1% of failures recover after a single attempt (five seconds). After four attempts (20 seconds), 75.3% of path failures recovered. Random-4 recovered from 60.2% of the failures by that point, while the path self-repaired from 15.1% of the failures. Since we abandon random-4 after four failed attempts, the fraction of paths that recover due to random-4 peaks at 60.2%. For broadband hosts, we see a similar pattern, except that random-4 is much less successful at recovering from failures: after four attempts, random-4 recovery has peaked at 29.4%.

Overall, we see that the major benefit of random-4 one-hop source routing is that it quickly finds alternative paths for recoverable failures. However, many failures are not recoverable with one-hop source routing. Bounding repeated random-4 to four attempts restricts the effort expended, instead allowing the path to self-repair.

3.9 Summary

This section examined specific policies for exploiting the potential benefit of one-hop source routing. Our results show that a simple, stateless policy called *random-4* comes close to obtaining the maximum gain possible. For example, from our trace data, random-4 found a successful intermediary for 60% of popular server failures (out of the 66% achievable shown previously in Section 2). Furthermore, random-4 is scalable and requires

no overhead messages. In comparison, the alternative knowledge-based policies we examined have higher cost and only limited benefit, relative to random-4.

The crucial question, then, is whether one-hop source routing can be implemented practically and can work effectively in a real system. We attempt to answer this question in the next section.

4 An Implementation Study

The goals of this section are twofold: (1) to discuss the pragmatic issues of integrating one-hop source routing into existing application and OS environments, and (2) to evaluate how well it works in practice. To this end, we developed a one-hop source routing prototype, which we call SOSR (for scalable one-hop source routing), and evaluate the prototype experimentally in the context of a Web-browsing application for popular servers.

4.1 Prototype Implementation

The high-level SOSR architecture consists of two major parts: one for the *source-node* and one for the *intermediary-node*. The source-node component retries failed communications from applications through one or more intermediaries. For scalability, all decision making rests with the source. The intermediary-node component forwards the messages it receives to the destination, acting as a proxy for the source. There is no destination-node component; SOSR is transparent to the destination.

Our source-node component, shown in Figure 15a, is implemented on top of the Linux *netfilter* framework [16]. Netfilter/iptables is a standard feature of the Linux kernel that simplifies construction of tools such as NATs, firewalls, and our SOSR system. It consists of an in-kernel module that forwards packets to a user-mode module for analysis. After receiving and examining a packet, the user-mode module can direct netfilter on how to handle it, acting in essence as an intelligent IP-level router.

We use the iptables rule-handling framework to inform netfilter about which packets to redirect to the user-level SOSR code and which to pass through. Our rules cause specific TCP flows (messages in both directions) to be redirected, based on port numbers or IP addresses. As well, they associate each flow with a SOSR policy module to handle its messages. Each policy module consists of failure detection code and intermediary selection code that decides when and where to redirect a packet.

The SOSR intermediary node acts as a NAT proxy [9], forwarding packets received from a source to the correct destination, and transparently forwarding packets from the destination back to the correct source. Figure 15b shows the encapsulation of indirected messages flowing through the intermediary. The source node encapsulates

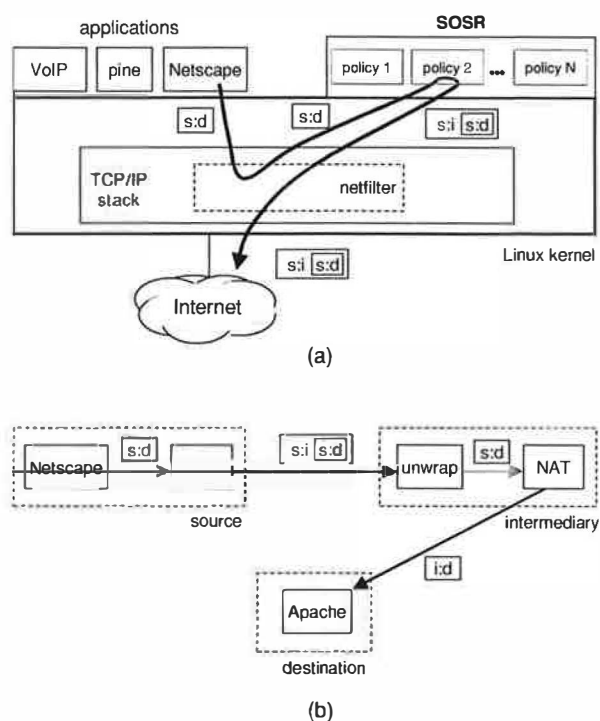


Figure 15: **The SOSR architecture.** (a) The source node uses netfilter to redirect packets from the Linux kernel to a user-level SOSR policy module, which encapsulates the application packet and tunnels it to a chosen intermediary. Note that the application is unmodified. (b) The intermediary node decapsulates the packet, passes it through a user-level NAT daemon, and transmits it to the (unmodified) destination node.

a TCP segment and information identifying the destination, and tunnels it to the intermediary through UDP. On receiving the UDP packet, the intermediary unwraps the TCP segment, passes it through its NAT component, and sends it to the destination using raw sockets. The NAT component of the intermediary maintains a hash table associating the sending node, the destination node, and the sending port it used to forward the segment. When a packet arrives back from the destination, the intermediary finds the associated (original) source address in the table, encapsulates the segment in a UDP packet along with the destination address, and returns it to the original source.

4.2 Evaluation Methodology

We evaluate our SOSR prototype by measuring its effectiveness in handling end-to-end Web-browsing failures for a set of popular Web servers. These failures include path failures, end-host failures, and application-level failures – anything that disrupts the browser from successfully completing an HTTP server transaction. The destination set consisted of 982 of the popular Web servers we considered in our measurement trace.

For our experiments, we used the same 39 SOSR intermediaries as in our earlier trace-based experiment. We then used three client machines at the University of Washington to repeatedly fetch Web pages from the destination set. Each machine ran a different error-handling protocol. The first simply ran the *wget* command-line Web browser on an unmodified Linux kernel. The second ran the same *wget* browser, but with the SOSR implementation in place, implementing the random-4 policy described in Section 3. The third ran *wget* on a modified, aggressive TCP stack; in place of standard TCP exponential back-off, we sent five duplicate packets every three seconds on a packet loss. This provides a balanced comparison to SOSR, in that it sends packets in the same number and frequency as random-4, but without the path diversity provided by the intermediaries.

Each client machine fetched a Web page once per second, rotating through the 982 Web servers a total of 279 times, ultimately issuing 273,978 requests. The machines fetched the same Web page at the same time, so that a path failure that affected one would hopefully affect the others as well. We ran our experiment for slightly more than 72 hours.

4.2.1 Failure classification

We classified failures as either *network-level failures* or *application-level failures*. If *wget* could not establish a TCP connection to the remote server, we classified the request as a network-level failure. However, if the destination returned a TCP RST packet to the source to refuse the connection, we classified the request as a “TCP refused” application-level failure, since the network path was clearly working.

If *wget* successfully established a TCP connection, but could not successfully complete an HTTP transaction within 15 minutes, we classified the request as an application-level failure. We further sub-classify application-level failures according to whether the HTTP transaction timed out (“HTTP timeout”) or was dropped by the server before completion (“HTTP refused”). Though we do not consider them to be failures, we also noted HTTP response that contained an HTTP error code, such as “HTTP/1.1 500 server error.”

4.3 Results

Table 4 summarizes our results. Overall, we observed few failures in our 72-hour experiment. The default *wget* client saw 481 failures in 273,978 requests, a failure rate of only 0.18%. We classified 69% as network-level failures and 31% as application-level failures. Thus, the network was responsible for approximately twice as many failures as the Web servers. However, indirect routing cannot recover from the Web server failures.

	requests	failures	network level failures	application level failures			HTTP error codes
				TCP refused	HTTP refused	HTTP timeout	
wget	273,978	481 (0.18%)	328 (0.12%)	40 (0.01%)	78 (0.03%)	35 (0.01%)	44 (0.02%)
wget SOSR	273,978	383 (0.14%)	145 (0.05%)	41 (0.01%)	101 (0.04%)	96 (0.04%)	37 (0.01%)
wget aggressiveTCP	273,978	486 (0.18%)	246 (0.09%)	43 (0.02%)	109 (0.04%)	88 (0.03%)	38 (0.01%)

Table 4: **Failures observed.** Summary of our SOSR prototype evaluation results, showing the number of network-level and application-level failures each of our three test clients observed.

In comparison, the *wget-SOSR* client experienced 20% fewer failures than the default *wget* client: 383 compared to 481. Of these, 145 (38%) were network-level failures and 238 (62%) were application-level failures. *wget-SOSR* recovered from many of the network-level failures that the default *wget* experienced. The network-level recovery rate for *wget-SOSR* was 56%, slightly below the 66% we measured in Section 3. However, since application-level errors are unrecoverable, *wget-SOSR*’s overall recovery rate was just 20%.

The *wget-aggressiveTCP* client experienced a marginally higher number of failures than the default *wget*: i.e., an aggressive TCP stack did not reduce the overall failure rate. However, it did reduce the number of network-level failures by 25%. The aggressive retransmissions were able to deal with some of the failures, but not as many as *wget-SOSR*. This confirms that SOSR derives its benefit in part because of path diversity, and in part because of its more aggressive retransmission.

Interestingly, both *wget-SOSR* and *wget-aggressiveTCP* observed a higher application-level failure rate than the default *wget*, receiving a significant increase in HTTP refused and timeout responses. While we cannot confirm the reason, we suspect that this is a result of the stress that the additional request traffic of these protocols causes on already overloaded servers. These additional application-level failures in balance reduced (for *wget-SOSR*) or canceled (for *wget-aggressiveTCP*) the benefits of the lower network-level failure rate.

4.4 Summary

This section presented a Linux-based implementation of one-hop source routing, called SOSR. SOSR builds on existing Linux infrastructure (netfilter/iptables) to route failures through a user-mode policy module on the source and a user-mode NAT proxy on intermediaries.

Our SOSR implementation was able to reduce recoverable (network-level) failures by 56% – close to what was predicted by our trace. However, with the Web-browsing application, we saw a new class of failures

caused by the Web servers themselves rather than the network; these application-level failures cannot be recovered using indirect network routing. Including these non-recoverable failures, our SOSR implementation was able to reduce the end-to-end failure rate experienced by Web clients by only 20%.

Given the extremely low failure rate that non-SOSR Web clients experience today, we do not believe that SOSR would lead to any noticeable improvement in a person's Web browsing experience. However, this does not imply that one-hop source routing is without value. SOSR is very successful at routing around non-last-hop network failures, and moreover, it has very little overhead. An application that requires better path availability than the Internet currently provides can achieve it using one-hop source routing, assuming that the paths it communicates over have relatively few last-hop failures. Finally, it is likely that SOSR achieves close to the maximum achievable benefit of alternative routing; overlay schemes that attempt to improve reliability are unlikely to better SOSR's recovery rate, and have significant scalability problems due to high message overhead as well.

5 Related Work

Internet reliability has been studied for at least a decade. Paxson's study of end-to-end paths found the likelihood of encountering a routing pathology to be 1.5% in 1994 and 3.3% in 1995 [17]. Zhang concluded that Internet routing had not improved five years later [26]. Chandra observed an average wide-area failure rate of 1.5%, close to "two 9s" of availability [5]. Labovitz's study of routing found path availability to vary widely between "one 9" and "four 9s" [13]. These studies all demonstrate that Internet reliability falls short of that measured for the telephone network [11].

Feamster characterized path failures between overlay nodes. He found wide variation in the quality of paths, failures in all regions of the network, and many short failures [7]. Our work offers a more comprehensive study of network failures that measures paths to a much larger set of Internet-wide destinations. Our results are largely consistent with these earlier findings.

Server availability may be improved using content distribution networks (CDNs) [10] and clusters [8, 4]. This is common for high-end Web sites. However, unlike our technique, it is applicable only to particular services, such as the Web.

Akella uses measurements to confirm that multi-homing has the potential to improve reliability as well as performance [1]. However, a strategy is still needed to select which path to use to obtain these benefits. Our technique is one such strategy; it will take advantage of multi-homing when it exists. There are also commercial

products that operate at the BGP routing level to select paths [19, 22]. The advantage of operating at the packet level, as we do, is more rapid response to failures. Conversely, it is known that BGP dynamics can result in a relatively long fail-over period [12] and that BGP misconfigurations are common [14].

Our work is most closely related to overlay routing systems that attempt to improve reliability and performance. The Detour study suggested that this could be accomplished by routing via intermediate end-systems [21]. RON demonstrated this to be the case in a small-scale overlay [2, 3]. Our work differs in two respects. First, we target general communication patterns rather than an overlay. This precludes background path monitoring. Second, and more fundamentally, we show that background monitoring is not necessary to achieve reliability gains. This eliminates overhead in the common no failure case. Our finding is consistent with the study by Teixeira [24] that observed a high-level of path diversity that is not exploited by routing protocols. The NATRON system [25] uses similar tunneling and NAT mechanisms as SOSR to extend the reach of a RON overlay to external, RON-oblivious hosts.

6 Conclusions

This paper proposed a simple and effective approach to recovering from Internet path failures. Our approach, called *one-hop source routing*, attempts to recover from path failures by routing indirectly through a small set of randomly chosen intermediaries. In comparison to related overlay-based solutions, one-hop source routing performs no background path monitoring, thereby avoiding scaling limits as well as overhead in the common case of no failure.

The ability to recover from failures only matters if failures occur in practice. We conducted a broad measurement study of Internet path failures by monitoring 3,153 randomly selected Internet destinations from 67 PlanetLab vantage points over a seven day period. We observed that paths have relatively high average availability (99.6% to popular servers, but only 94.4% for broadband), and that most monitored paths experienced at least one failure. However, 16% of failures on paths to servers and 60% of failures on paths to broadband hosts were located on the last-hop or end-host. It is impossible to route around such last-hop failures. Overall, our measurement study demonstrated that a simple one-hop source routing technique called "random-4" could recover from 61% of path failures to popular servers, but only 35% of path failures to broadband hosts.

We implemented and deployed a prototype one-hop source routing infrastructure on PlanetLab. Over a 48 hour period, we repeatedly accessed 982 popular Web servers and used one-hop source routing to attempt to

route around failures that we observed. Our prototype was able to recover from 56% of network failures, but we also observed a large number of server failures that cannot be addressed through alternative routing techniques. Including such application-level failures, our prototype was able to recover from 20% of failures encountered.

In summary, one-hop source routing is easy to implement, adds negligible overhead, and achieves close to the maximum benefit available to any alternative routing scheme, *without* the need for path monitoring, history, or a-priori knowledge of any kind.

7 Acknowledgments

We wish to thank Brian Youngstrom and the members of PlanetLab support who helped us with our trace, and Stavan Parikh who helped us develop SOSR. We also gratefully acknowledge the valuable feedback of Neil Spring, Ratul Mahajan, and Marianne Shaw, and we thank Miguel Castro for serving as our shepherd. This research was supported by the National Science Foundation under Grants ITR-0121341 and CCR-0085670 and by a gift from Intel Corporation.

References

- [1] A. Akella, B. Maggs, S. Seshan, A. Shaikh, and R. Sitaraman. A measurement-based analysis of multihoming. In *Proceedings of ACM SIGCOMM 2003*, Karlsruhe, Germany, August 2003.
- [2] D. G. Andersen, N. Feamster, S. Bauer, and H. Balakrishnan. Resilient overlay networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Marseille, France, November 2002.
- [3] D. G. Andersen, A. C. Snoeren, and H. Balakrishnan. Best-path vs. multi-path overlay routing. In *Proceedings of ACM/USENIX Internet Measurement Conference 2003*, October 2003.
- [4] V. Cardellini, E. Casalicchio, M. Colajanni, and P. S. Yu. The state of the art in locally distributed web-server systems. Technical Report RC22209 (W0110-048), IBM T. J. Watson Research Center, October 2001.
- [5] M. Dahlin, B. Chandra, L. Gao, and A. Nayate. End-to-end wan service availability. *IEEE/ACM Transactions on Networking*, 11(2), April 2003.
- [6] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, 12(5):662–675, May 1986.
- [7] N. Feamster, D. Andersen, H. Balakrishnan, and F. Kaashoek. Measuring the effects of internet path faults on reactive routing. In *Proceedings of ACM SIGMETRICS 2003*, San Diego, CA, June 2003.
- [8] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 78–91, Saint Malo, France, 1997.
- [9] Network Working Group. The ip network address translator (nat). RFC 1631, May 1994.
- [10] B. Krishnamurthy, C. Willis, and Y. Zhang. On the use and performance of content distribution networks. In *Proceedings of ACM SIGCOMM Internet Measurement Workshop 2001*, November 2001.
- [11] D. R. Kuhn. Sources of failure in the public switched telephone networks. *IEEE Computer*, 30(4):31–36, April 1997.
- [12] C. Labovitz, A. Ahuja, A. Abose, and F. Jahanian. An experimental study of delayed internet routing convergence. In *Proceedings of ACM SIGCOMM 2000*, Stockholm, Sweden, August 2000.
- [13] C. Labovitz, A. Ahuja, and F. Jahanian. Experimental study of internet stability and backbone failures. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing*, pages 278–285, 1999.
- [14] R. Mahajan, D. Wetherall, and T. Anderson. Understanding bgp misconfiguration. In *Proceedings of ACM SIGCOMM 2002*, pages 3–16, Pittsburgh, PA, 2002.
- [15] M. Mitzenmacher. On the analysis of randomized load balancing schemes. Technical Report 1998-001, Digital Systems Research Center, February 1998.
- [16] Netfilter. <http://www.netfilter.org>.
- [17] V. Paxson. End-to-end routing behavior in the internet. *IEEE/ACM Transactions on Networking*, 5(5):601–615, October 1997.
- [18] PlanetLab. <http://www.planet-lab.org>.
- [19] RouteScience. <http://www.routescience.com>.
- [20] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking (MMCN) 2002*, January 2002.
- [21] S. Savage, A. Collins, E. Hoffman, J. Snell, and T. Anderson. The end-to-end effects of Internet path selection. In *Proceedings of ACM SIGCOMM 1999*, Cambridge, MA, September 1999.
- [22] Sockeye. <http://www.sockeye.com>.
- [23] L. Subramaniam, S. Agarwal, J. Rexford, and R. H. Katz. Characterizing the internet hierarchy from multiple vantage points. In *Proceedings of IEEE INFOCOM 2002*, New York, USA, June 2002.
- [24] R. Teixeira, K. Marzullo, S. Savage, and G. M. Voelker. Characterizing and measuring path diversity of internet topologies. In *Proceedings of ACM SIGMETRICS 2003*, pages 304–305, San Diego, CA, 2003.
- [25] Alexander Siumann Yip. NATRON: overlay routing to oblivious destinations. Master's thesis, 2002.
- [26] Y. Zhang, V. Paxson, and S. Shenker. The stationarity of internet path properties: Routing, loss, and throughput. Technical report, ACIRI, May 2000.

CoDNS: Improving DNS Performance and Reliability via Cooperative Lookups

KyoungSoo Park, Vivek S. Pai, Larry Peterson and Zhe Wang
Department of Computer Science
Princeton University

Abstract

The Domain Name System (DNS) is a ubiquitous part of everyday computing, translating human-friendly machine names to numeric IP addresses. Most DNS research has focused on server-side infrastructure, with the assumption that the aggressive caching and redundancy on the client side are sufficient. However, through systematic monitoring, we find that client-side DNS failures are widespread and frequent, degrading DNS performance and reliability.

We introduce CoDNS, a lightweight, cooperative DNS lookup service that can be independently and incrementally deployed to augment existing nameservers. It uses a locality and proximity-aware design to distribute DNS requests, and achieves low-latency, low-overhead name resolution, even in the presence of local DNS nameserver delay/failure. Using live traffic, we show that CoDNS reduces average lookup latency by 27-82%, greatly reduces slow lookups, and improves DNS availability by an additional '9'. We also show that a widely-deployed service using CoDNS gains increased capacity, higher reliability, and faster start times.

1 Introduction

The Domain Name System (DNS) [15] has become a ubiquitous part of everyday computing due to its effectiveness, human-friendliness, and scalability. It provides a distributed lookup service primarily used to convert from human-readable machine names to Internet Protocol (IP) addresses. Its existence has permeated much of computing via the World Wide Web's near-complete dependence on it. Thanks in part to its redundant design, aggressive caching, and flexibility, it has become a ubiquitous part of everyday computing that most people take for granted, including researchers.

Most DNS research focuses on "server-side" problems, which arise on the systems that translate names belonging to the group that runs them. Such problems include understanding name hierarchy misconfiguration [5, 9] and devising more scalable distribution infrastructure [4, 10, 18]. However, due to increasing memory sizes and DNS's high cachability, "client-side" DNS hit rates are approaching 90% [9, 24], so fewer requests are dependent on server-side performance. The

client-side components are responsible for contacting the appropriate servers, if necessary, to resolve any name presented by the user. This infrastructure, which has received less attention, is our focus – understanding client-side behavior in order to improve overall DNS performance and reliability.

Using PlanetLab [16], a wide-area distributed testbed, we locally monitor the client-side DNS infrastructure of 150 sites around the world, generating a large-scale examination of client-side DNS performance. We find that client-side failures are widespread and frequent, and that their effects degrade DNS performance and reliability. The most common problems we observe are intermittent failures to receive any response from the local nameservers, but these are generally hidden by the internal redundancy in DNS deployments. However, the cost of such redundancy is additional delay, and we find that the delays induced through such failures often dominate the time spent waiting on DNS lookups.

To address these client-side problems, we have developed CoDNS, a lightweight, cooperative DNS lookup service that can be independently and incrementally deployed to augment existing nameservers. CoDNS uses an insurance-like model of operation – groups of mutually trusting nodes agree to resolve each other's queries when their local infrastructure is failing. We find that the group size does not need to be large to provide substantial benefits – groups of size 2 provide roughly half the maximum possible benefit, and groups of size 10 achieve almost all of the possible benefit. Using locality-enhancement techniques and proximity optimizations, CoDNS achieves low-latency, low-overhead name resolution, even in the presence of local DNS delays/failures.

CoDNS has been serving live traffic on PlanetLab since October 2003, providing many benefits over standard DNS. CoDNS reduces average lookup latency by 27-82%, greatly reduces slow lookups, and improves DNS availability by an extra '9', from 99% to over 99.9%. Its service is more reliable and consistent than any individual node's. Additionally, CoDNS has salvaged "unusable" nodes, which had such poor local DNS infrastructure that they were unfit for normal use. Applications using CoDNS often have faster and more predictable start times, improving availability.

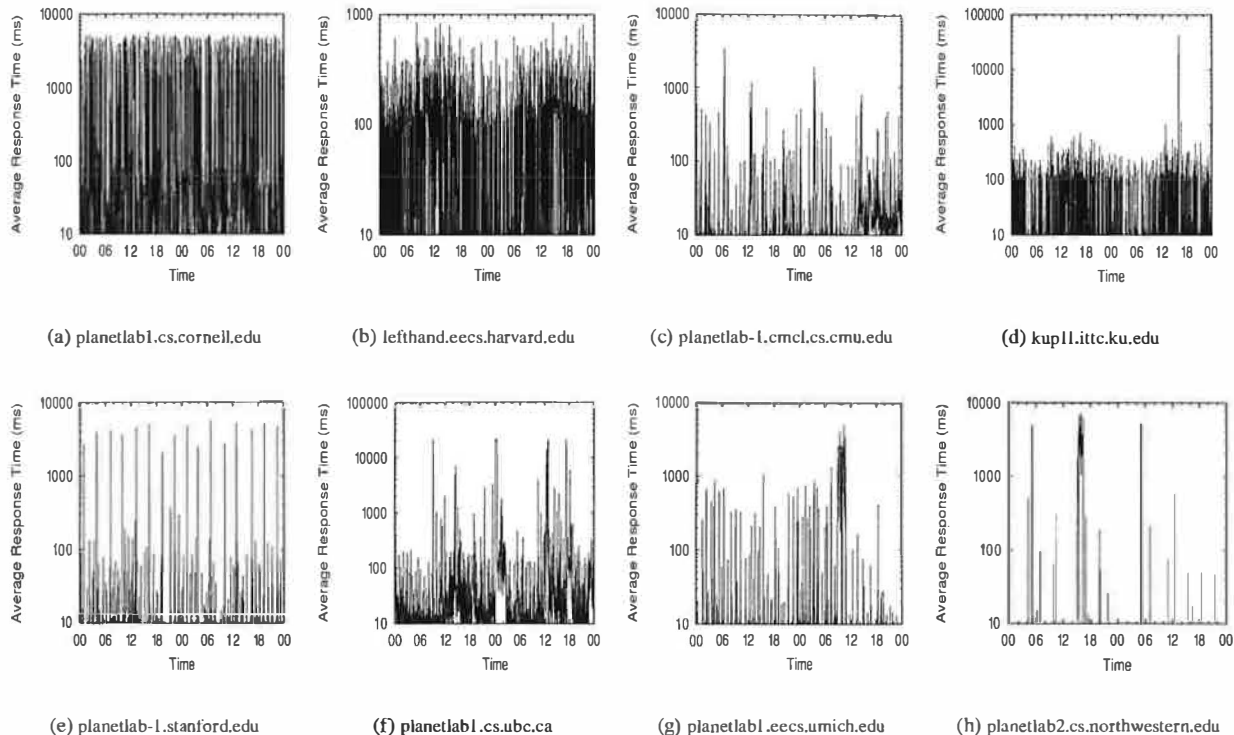


Figure 1: Average cached DNS lookup response times on various PlanetLab nodes over two days. Note that while most Y axes span 10-1000 milliseconds, some are as large as 100,000 milliseconds.

2 Background & Analysis

While the Domain Name System (DNS) was intended to be a scalable, distributed means of performing name-to-IP mappings, its flexible design has allowed it to grow far beyond its original goals. While most people would be familiar with it for Web browsing, many systems depend on fast and consistent DNS performance. Mail servers, Web proxy servers, and content distribution networks (CDNs) must all resolve hundreds or even thousands of DNS names in short periods of time, and a failure in DNS may cause a service failure, rather just delays.

The server-side infrastructure of DNS consists of hierarchically-organized name servers, with central authorities providing “root” servers and others delegated organizations handling “top-level” servers, such as “.com” and “.edu”. Domain name owners are responsible for providing servers that handle queries for their names. While DNS users can manually query each level of the hierarchy in turn until the complete name has been resolved, most systems delegate this task to local nameserver machines. This approach has performance advantages (e.g., caching replies, consolidating requests) as well as management benefits (e.g., fewer machines to update with new software or root server lists).

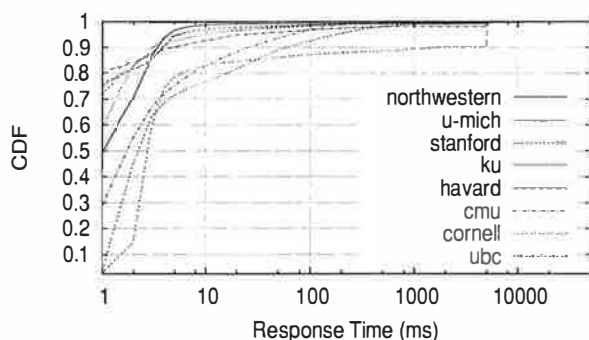
With local nameserver cache hit rates approaching 90% [9, 24], their performance impact can eclipse that

of the server-side DNS infrastructure. However, local nameserver performance and reliability has not been well studied, and since it handles all DNS lookups for clients, its failure can disable other systems. Our experiences with building the CoDeeN content distribution network, running on over 100 PlanetLab nodes [23], motivated us to investigate this issue, since all CoDeeN nodes use the local nameservers at their hosting sites.

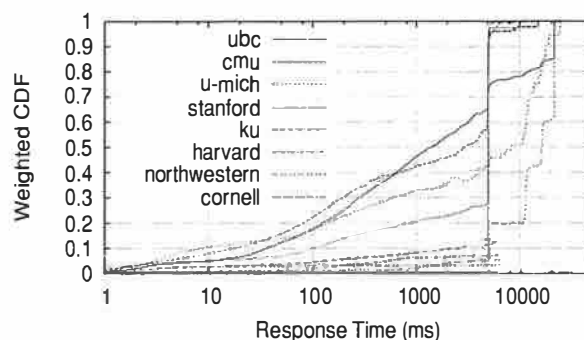
2.1 Frequency of Name Lookup Failures

To determine the failure properties of local DNS infrastructure, we systematically measure DNS lookup times on many PlanetLab nodes. In particular, across 40 North American sites, we perform a query once per second. We ask these nodes to resolve each other’s names, all of which are cacheable, with long time-to-live (TTL) values of no less than 6 hours. Lookup times for these requests should be minimal, on the order of a few milliseconds, since they can be served from the local nameserver’s cache. This diagnostic workload is chosen precisely because it is trivially cacheable, making local infrastructure failures more visible and quantifiable. Evaluation of DNS performance on live traffic, with and without CoDNS, is covered in Section 5.

Our measurements show that local DNS lookup times are generally good, but often degrade dramatically, and that this instability is widespread and frequent. To illus-



(a) Fraction of Lookups Taking < X ms



(b) Fraction of the Sum of Lookups Taking < X ms

Figure 2: Cumulative Distribution of Cached DNS Lookups

trate the widespread nature of the problem and its magnitude, Figure 1 shows the lookup behavior over a two-day period across a number of PlanetLab nodes. Each point shows the per-minute average response time of name lookups. All the nodes in the graph show some sort of problems in DNS lookups during the period, with lookups often taking thousands of milliseconds.

These problems are not consistent with simple configuration problems, but appear to be usage-induced or triggered by activity on the nameserver nodes. For example, the Cornell node consistently shows DNS problems, with more than 20% of lookups showing high lookup times of over five seconds, the default timeout in the client's resolver library. These failed lookups are eventually retried at the campus's second nameserver, masking the first nameserver's failures. Since the first nameserver responds to 80% of queries in a timely manner, it is not completely misconfigured. Very often throughout the day, it simply stops responding, driving the per-minute average lookup times close to five seconds. The Harvard node also displays generally bad behavior. While most lookups are fine, a few failed requests every minute substantially increase the per-minute average. The Stanford node's graph shows periodic spikes roughly every three hours. This phenomenon is long-term, and we suspect the nameserver is being affected by heavy cron jobs. The Michigan node shows a 90 minute DNS problem, driving its generally low lookup times to above one second.

Although the average lookup times appear quite high at times, the individual lookups are mostly fast, with a few very slow lookups dominating the averages. Figure 2(a) displays the cumulative distribution function (CDF) of name lookup times over the same two days. With the exception of the Cornell node, 90% of all requests take less than 100ms on all nodes, indicating that caching is effective and that average-case latencies are quite low. Even the Cornell node works well most of the time, with over 80% of lookups are resolved within 6ms.

Node	Avg	Low	High	T-Low	T-High
cornell	531.7ms	82.4%	12.9%	0.5%	99.2%
harvard	99.4ms	92.3%	3.3%	0.7%	97.9%
cmu	24.0ms	81.9%	3.2%	8.3%	71.0%
ku	53.1ms	94.6%	1.8%	2.9%	95.0%
stanford	21.5ms	95.7%	1.3%	5.3%	89.5%
ubc	88.8ms	76.0%	7.6%	2.4%	91.2%
umich	43.6ms	96.7%	1.3%	2.4%	96.1%
northwestern	43.1ms	98.5%	0.5%	4.5%	94.8%

Table 1: Statistics over two days, Avg = Average, Low = Percentage of lookups < 10 ms, High = Percentage of lookups > 100 ms, T-Low = Percentage of total low time, T-High = Percentage of total high time

However, slow lookups dominate the total time spent waiting on DNS, and are large enough to be noticeable by end users. In Figure 2(b), we see the lookups shown by their contribution to the total lookup time, which indicates that **a small percentage of failure cases dominates the total time**. This weighted CDF shows, for example, that none of the nodes crosses the 0.5 value before 1000ms, indicating that more than 50% of the lookup time is spent on lookups taking more than 1000ms. If we assume that a well-behaving local nameserver can serve cached responses in 100ms, then the figures are even more dramatic. This data, shown in Table 1, shows that slow lookups comprise most of the lookup time.

These measurements show that **client-side DNS infrastructure problems are significant** and need to be addressed. If we can reduce the amount of time spent on these longer cases, particularly in the failures that require the local resolver to retry the request, we can dramatically reduce the total lookup times. Furthermore, given the sharp difference between "good" and "bad" lookups, we may also be able to ensure a more predictable (and hence less annoying) user experience. Finally, it is worth noting that these problems are not an artifact of PlanetLab – in all cases, we use the site's local nameservers, on which hundreds or thousands of other non-PlanetLab

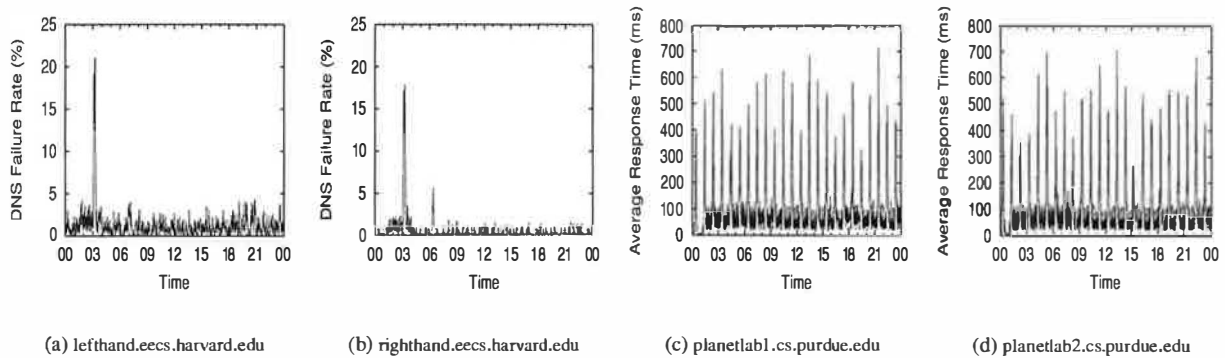


Figure 3: All nodes at a site see similar local DNS behavior, despite different workloads at the nodes. Shown above are one day's failure rates at Harvard, and one day's response times at Purdue.

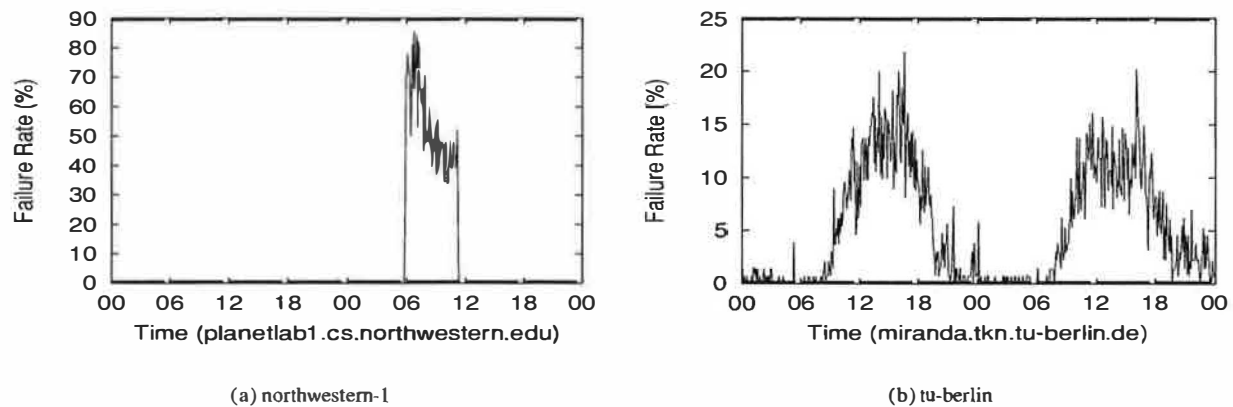


Figure 4: Failures seemingly caused by nameserver overload – in both cases, the failure rate is always less than 100%, indicating that the server is operational, but performing poorly.

machines depend. The PlanetLab nodes at a site see similar lookup times and failure rates, despite the fact that their other workloads may be very different. Examples from two sites are shown in Figure 3, and we can see that the nodes at a site see similar DNS performance. This observation further enhances our claim that the problems are site-wide, and not PlanetLab-specific.

2.2 Origins of the Client-Side Failures

While we do not have full access to all of the client-side infrastructure, we can try to infer the reasons for the kinds of failures we are seeing and understand their impact on lookup behavior. Absolute confirmation of the failure origins would require direct access to the nameservers, routers, and switches at the sites, which we do not have. Using various techniques, we can trace some problems to packet loss, nameserver overloading, resource competition and maintenance issues. We discuss these below.

Packet Loss – The simplest cause we can guess is the packet loss in the LAN environment. Most nameservers communicate using UDP, so even a single packet loss either as a request or as a response would eventually trigger

a query retransmission from the resolver. The resolver's default timeout for retransmission is five seconds, which matches some of the spikes in Figure 1.

Packet loss rates in LAN environments are generally assumed to be minimal, and our measurements of Princeton's LAN support this assumption. We saw no packet loss at two hops, 0.02% loss at three hops, and 0.09% at four hops. Though we did see bursty behavior in the loss rate, where the loss rates would stay high for a minute at a time, we do not see enough losses to account for the DNS failures. Our measurements show that 90% of PlanetLab nodes have a nameserver within 4 hops, and 70% are within 2 hops. However, other contexts, such as cable modems or dial-up services, have more hops [20], and may have higher loss rates.

Nameserver overloading – Since most request packets are likely to reach the nameserver, our next possible culprit is the nameserver itself. To understand their behavior, we asked all nameservers on PlanetLab to resolve a local name once every two seconds and we measured the results. For example, on planetlab-1.cs.princeton.edu, we asked for planetlab-2.cs.princeton.edu's IP address.

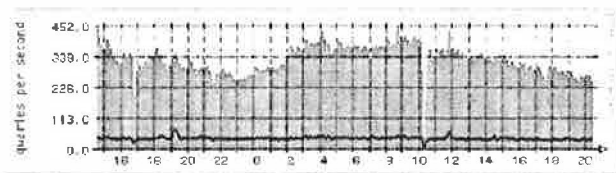


Figure 5: Daily Request Rate for Princeton.EDU

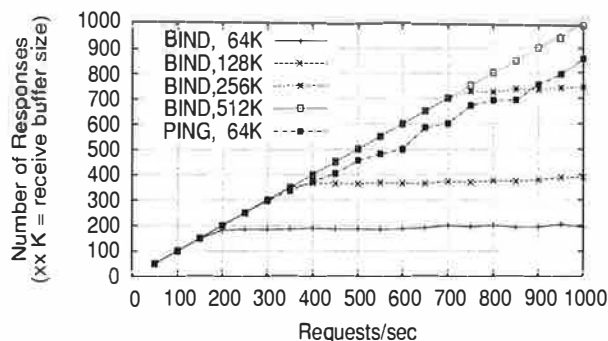


Figure 6: BIND 9.2.3 vs. PING with bursty traffic

In addition to the possibility of caching, the local nameserver is mostly likely the authoritative nameserver for the queried name, or at least the authoritative server can be found on the same local network.

In Figure 4, we see some evidence that nameservers can be temporarily overloaded. These graphs cover two days of traffic, and show the 5-minute average failure rate, where a failure is either a response taking more than five seconds, or no response at all. In Figure 4(a), the node experiences no failures most of time but a 30% to 80% failure rate for about five hours. Figure 4(b) reveals a site where failures start during the start of the workday, gradually increase, and drop starting in the evening. It is reasonable to assume that human activity increases in these hours, and affects the failure rate.

We suspect that a possible factor in this overloading is the UDP receive buffer on the nameserver. These buffers are typically sized in the range of 32-64KB, and incoming packets are silently dropped when this buffer is full. If the same buffer is also used to receive the responses from other nameservers, as the BIND nameserver does, this problem gets worse. Assuming a 64KB receive buffer, a 64 byte query, and a 300 byte response, more than 250 simultaneous queries can cause packet dropping. In Figure 5, we see the request rate (averaged over 5 minutes) for the authoritative nameserver for princeton.edu. Even with smoothing, the request rates are in the range of 250-400 reqs/sec, and we can expect that instantaneous rates are even higher. So, any activity that causes a 1-2 second delay of the server can cause requests to be dropped.

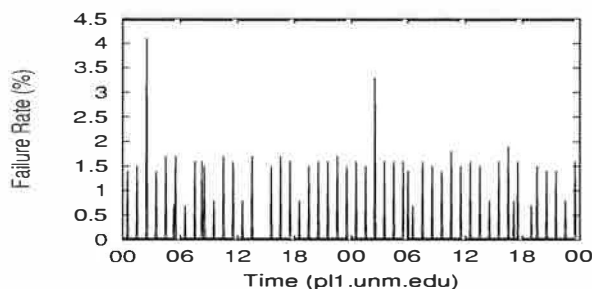


Figure 7: This site shows failures induced by periodic activity. In addition to the hourly failure spike, a larger failure spike is seen once per day.

To test this theory of nameserver overload, we subjected BIND, the most popular nameserver, to bursty traffic. On an otherwise unloaded box (Compaq au600, Linux 2.4.9, 1 GB memory), we ran BIND 9.2.3 and an application-level UDP ping that simulates BIND. Each request contains the same name query for a local domain name with a different query ID. Our UDP ping responds to it by sending a fixed response with the same size as BIND's. We send a burst of N requests from a client machine and wait 10 seconds to gather responses. Figure 6 shows the difference in responses between BIND 9.2.3 and our UDP ping. With the default receive buffer size of 64KB, BIND starts dropping requests at bursts of 200 reqs/sec, and the capacity linearly grows with the size of the receive buffer. Our UDP ping using the default buffer loses some requests due to temporary overflow, but the graph does not flatten because responses consume minimal CPU cycles. These experiments confirm that high-rate bursty traffic can cause server overload, aggravating the buffer overflow problem.

Resource competition – Some sites show periodic failures, similar to what is seen in Figure 7. These tend to have spikes every hour or every few hours, and suggests some heavy process is being launched from cron. BIND is particularly susceptible to memory pressure, since its memory cache is only periodically flushed. Any jobs that use large amounts of memory can evict BIND's pages, causing BIND to page fault when accessing the data. The faults can delay the server, causing the UDP buffer to fill.

In talking with system administrators, we find that even sites with good DNS service often run multiple services (some cron-initiated) on the same machine. Since DNS is regarded as a low-CPU service, other services are run on the same hardware to avoid underutilization. It seems quite common that when these other services have bursty resource behavior, the nameserver is affected.

Maintenance problems – Another common source of failure is maintenance problems which lead to service interruption, as shown in Figure 8. Here, the DNS lookup shows a 100% failure rate for 13 hours. Both name-

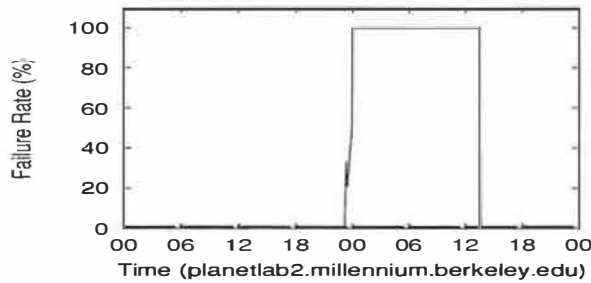


Figure 8: This site’s nameservers were shut down before the nodes had been updated with the new nameserver information. The result was a 13-hour complete failure of all name lookups, until the information was manually updated.

servers for this site stopped working causing DNS to be completely unavailable, instead of just slow. DNS service was restored only after manual intervention. Another common case, complete failure of the primary nameserver, generates a similar pattern, with all responses being retried after five seconds and sent to the secondary nameserver.

3 Design

In this section, we discuss the design of CoDNS, a name lookup system that provides faster and more reliable DNS service while minimizing extra overhead. We also discuss the observations that shape this approach. Using trace-driven workloads, we calculate the overheads and benefits of various design choices in the system.

One important goal shapes our design: our system should be incrementally deployable, not only by DNS administrators, but also by individual users. The main reason for this decision is that it bypasses the bureaucratic processes involved with replacing existing DNS infrastructure. Given the difficulty we have in even getting information about local DNS nameservers, the chances of convincing system administrators to send their live traffic to an experimental name lookup service seems low. Providing a migration path that coexists with existing infrastructure allows people the opportunity to grow comfortable with the service over time.

Another implication of this strategy is that we should aim for minimal resource commitments. In particular, we should leverage the existing infrastructure devoted to making DNS performance generally quite good. Client-side nameservers achieve high cache hit rates by devoting memory to name caching, and if we can take advantage of the existing infrastructure, it lessens the cost of deployment. While current client-side infrastructure, including nameservers, is not perfect, it provides good performance most of the time, and it can provide a useful starting point. Low resource usage also reduces the chances for failure due to resource contention.

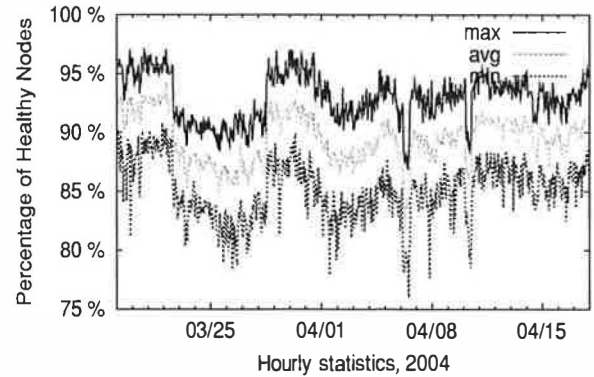


Figure 9: Hourly % of nodes with working nameservers

Our usage model is cooperative, operating similarly to insurance – nodes join a pool that shares resources in times of need. If a node’s local lookup performance is acceptable, it proceeds as usual, but may have to provide service to nodes that are having problems. When its local performance degrades, it can ask other nodes to help it. The benefit of joining is the ability to get help when needed, even if there is some overhead at other times.

3.1 Cross-site Correlation of DNS Failures

The “insurance” model depends on failure being relatively uncorrelated – the system must always have a sufficient pool of working participants to help those having trouble. If failure across sites is correlated, this assumption is violated, and a cooperative lookup scheme is less feasible. To test our assumption, we study the correlation of DNS lookup failures across PlanetLab. At every minute, we record how many nodes have “healthy” DNS performance. We define healthy as showing no failures for one minute for the local domain name lookup test. Using the per-minute data for March 2004, we show the minimum, average and maximum number of nodes available per hour. The percentage of healthy nodes (as a fraction of live nodes) is shown in Figure 9.

From this graph, we can see some minor correlation in failures, shown as downward spikes in the percentage of available nodes, but most of the variation in availability seems largely uncorrelated. An investigation into the spikes reveals that many nodes on PlanetLab are configured to use the same set of nameservers, especially those colocated at Internet2 backbone facilities (not to be confused with Internet2-connected university sites). When these nameservers experience problems, the correlation appears large due to the number of nodes affected.

More important, however, is the observation that the fraction of healthy nameservers is always high, generally above 90%. This observation provides the key insight for CoDNS – with enough healthy nameservers, we can mask locally-observed delays via cooperation.

Software	PlanetLab	Packetfactory	TLD
BIND-4.9.3+/8	31.1%	36.4%	55.9%
BIND 9	48.9%	25.1%	34.0%
Other	20.0%	38.5%	10.1%

Table 2: Comparison of nameserver software used by PlanetLab, packetfactory survey and the TLD survey

To ensure that these failures are not tied to any specific nameserver software, we survey the software running on the local nameservers used by the PlanetLab nodes (135 unique nameservers) with “chaos” class queries [14]. We find that they are mostly running a variety of BIND versions. We observe 11 different BIND 9 version strings, 13 different BIND 8 version strings and a number of humorous strings (included in “other”) apparently set by the nameserver administrators. These measurements, shown in Table 2, are in line with two recent nameserver surveys conducted by Brad Knowles in 2002 [11] and by packetfactory in 2003 [19]. From this, we conclude that the failures are not likely to be specific to PlanetLab’s choices of nameserver software.

3.2 CoDNS

The main idea behind CoDNS is to forward name lookup queries to peer nodes when the local name service is experiencing a problem. Essentially, this strategy applies a CDN approach to DNS – spreading the load among peers improves the size and performance of the “global cache”. Many of the considerations in CDN systems apply in this environment. We need to consider the proximity and availability of a node as well as the locality of the queries. A different consideration is that we need to decide when it is desirable to send remote queries. Given the fact that most name lookups are fast in the local nameserver, simply spreading the requests to peers might generate unnecessary traffic with no gain in latency. Worse, the extra load may cause marginal DNS nameservers to become overloaded. We investigate considerations for deciding when to send remote queries, how many peers to involve, and what sorts of gains to expect.

To precisely determine the effects of locality, load, and proximity is difficult, since we have no control over the nameservers and have little information about their workloads, configurations, etc. The proximity of a peer server is important in that DNS response time can be affected by its peer to peer latency. Since the DNS requests and responses are not large, we are more interested in picking nearby peers with low round-trip latency instead of nodes with particularly high bandwidth. We have observed coast-to-coast round-trip ping times of 80ms in CoDeeN, with regional times in the 20ms range. Both of these ranges are much lower than the DNS timeout value of five seconds, so, in theory, any node would be an acceptable peer. In practice, choosing closer peers will reduce the difference between cache hit times and remote

peer times, making CoDNS failure masking more transparent. For request locality, we would like to increase the chances of remote queries being cache hits in the remote nameservers. Using any scheme that consistently partitions this workload will help reduce cache pollution, and increase the likelihood of cache hits.

To understand the relationship between CoDNS response times, the number of peers involved, and the policies for determining when requests should be sent remotely, we collected 44,486 unique hostnames from one day’s HTTP traffic on CoDeeN and simulated various policies and their effects. We replayed DNS lookups of those names at 77 PlanetLab nodes with different nameservers, starting requests at the same time of day in the original logs. The replay happened one month after the data collections to avoid local nameserver caches which could skew the data. During this time, we also use application-level heartbeat measurements between all pairs of nodes to determine their round-trip latencies. Since all of the nodes are doing DNS lookups at about the same time, by adding the response time at peerY to the time spent for the heartbeat from peerX to peerY, we will get the response time peerX can get if it asks peerY for a remote DNS lookup for the same hostname.

An interesting question is how many simultaneous lookups are needed to achieve a given average response time and to reduce the total time spent on slow lookups (defined as taking more than 1 second). As shown in the previous section, it is desirable to reduce the number of slow responses to reduce the total lookup time. Figures 10 and 11 show two graphs answering this question. The lookup scheme here is to contact the local nameserver first for a name lookup, wait for a timeout and issue $x-1$ simultaneous lookups using $x-1$ randomly-selected peer nodes. Figures 10 shows that even if we use only one extra lookup, we can reduce the average response time by more than half. Also, beyond about five peers, adding more simultaneous lookups produces diminishing returns. Different initial timeout values do not produce much difference in response times, because the benefit largely stems from reducing the number of slow lookups. The slow response portion graph proves this phenomenon, showing similar reduction in the slow response percentage at any initial timeout less than 700ms.

We must also consider the extra overhead of the simultaneous lookups, since shorter initial timeouts and more simultaneous lookups causes more DNS traffic at all peers. Figure 12 shows the overhead in terms of extra lookups needed for various scenarios. Most curves start to flatten at a 500ms initial timeout, providing only diminishing returns for larger timeouts. Worth noting is that even with one peer and a 200ms initial timeout, we can still cut the average response time by more than half, with only 38% extra DNS lookups.

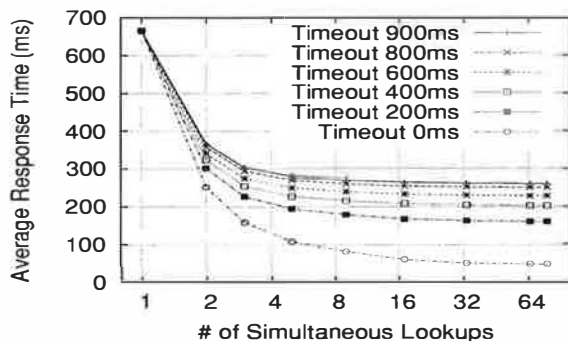


Figure 10: Average Response Time

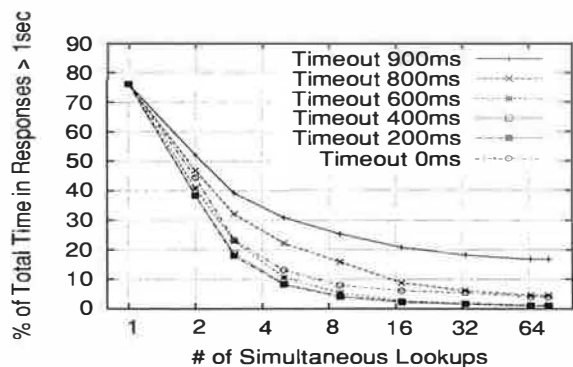


Figure 11: Slow Response Time Portion

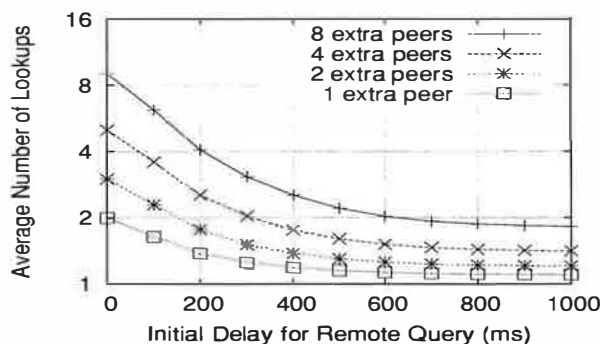


Figure 12: Extra DNS Lookups

These results are very encouraging, demonstrating that CoDNS can be effective even at very small scale – even a single extra site provides significant benefits, and it achieves most of its benefits with less than 10 sites. The reasons for this scale being important is twofold: only small commitments are required to try a CoDNS deployment, and DNS’s limitations with respect to trust and verification (discussed in the next section) are unlikely to be an issue at these scales.

3.3 Trust, Verification, and Implications

Some aspects of DNS and its design invariably impact our approach, and the most important is trust and verification. The central issue is whether it is possible for a requestor to determine that its peer has correctly resolved the request, and that the result provided is actually a valid IP address for the given name. This issue arises if peers can be compromised or are otherwise failing.

Unfortunately, we believe that a general solution to this problem is not possible with the current DNS, though certain fault models are more amenable to checking than others. For example, if the security model assumes that at most one peer can be compromised, it may be possible to always send remote requests to at least three peers. When these nodes respond, if two results agree, then the answer must be correct. However, DNS does not man-

date that any of these results have to agree, making the general case of verification impossible.

Many server-side DNS deployments use techniques to improve performance, reliability, or balance load and locality. For example, round-robin DNS can return results from a list of IP addresses in order to distribute load across a set of servers. Geography-based redirection can be used to reduce round-trip times between clients and servers by having DNS lookups resolve to closer servers. Finally, DNS-based content distribution networks will often incorporate load balancing and locality considerations when resolving their DNS names. In these cases, multiple lookups may produce different results, and lookups from different locations may receive results from different pools of IP addresses.

While it would be possible to imagine extending DNS such that each name is associated with a public key, and each IP address result is signed with this key, such a change would be significant. DNSSEC [6] attempts smaller-scale change, mainly to prevent DNS spoofing, but has been in some form of development for nearly a decade, and still has not seen wide-scale adoption.

Given the current impossibility of verifying all lookups, we rely on trusting peers in order to sidestep the problems mentioned. This approach is already used in various schemes. Name owners often use each other as their secondary servers, sometimes at large scale. For example, princeton.edu’s DNS servers act as the secondary servers for 60 non-Princeton domains. BIND supports zone transfers, where all DNS information can be downloaded from another node, specifically for this kind of scenario. Similarly, large-scale distributed systems running at hosting centers already have a trust relationship in place with their hosting facility.

4 Implementation

We have built a prototype of CoDNS and have been running it on all nodes on PlanetLab for roughly eight months. During that time, we have been directing the CoDeeN CDN [23] to use CoDNS for the name lookup.

CoDNS consists of a stand-alone daemon running on each node, accessible via UDP for remote queries, and via loopback TCP for locally-originated name lookups. The daemon is event-driven, and is implemented as a non-blocking master process and many (blocking) slave processes. The master process receives name lookup requests from local clients and remote peers, and passes them to one of its idle slaves. A slave process resolves those names by calling `gethostbyname()` and sends the result back to the master. Then, the master returns the final result to either a local client or a remote peer depending on where it originated. Queries resolving the same hostname are coalesced into one query and answered together when resolved. Preference for idle slaves is given to locally-originated requests over remote queries to ensure good performance for local users.

The master process records each request's arrival time from local clients and sends a UDP name lookup query to a peer node when the response from the slave has not returned within a certain period. This delay is used as a boundary for deciding if the local nameserver is slow. In the event that neither the local nameserver nor the remote node has responded, CoDNS doubles the delay value before sending the next remote query to another peer. In the process, whichever result that comes first will be delivered as the response for the name lookup to the client. Peers may silently drop remote queries if they are overloaded, and remote queries that fail to resolve are also discarded. Slaves may add delay if they receive a locally-generated request that fails to resolve, with the hope that remote nodes may be able to resolve such names.

4.1 Remote Query Initiation & Retries

The initial delay before sending the first remote query is dynamically adjusted based on the recent performance of local nameservers and peer responses. In general, when the local nameserver performs well, we increase the delay so that fewer remote queries are sent. When most remote answers beat the local ones, we reduce the delay preferring the remote source. Specifically, if the past 32 name lookups are all resolved locally without using any remote queries, then the initial delay is set to 200ms by default. We choose 200ms because the median response time on a well-functioning node is less than 100ms [9], so 200ms delay should respond fast during instability, while wasting minimal amount of extra remote queries.

However, to respond quickly to local nameserver failure, if the remote query wins more than 50% of the last 16 requests, then the delay is set to 0 ms. That is, the remote query is sent immediately as the request arrives. Our test results show it is rare not to have failure when more than 8 out of 16 requests take more than 300ms to resolve, so we think it is reasonable to believe the local nameserver is having a problem in that case. Once the immediate query is sent, the delay is set to the average

response time of remote query responses plus one standard deviation, to avoid swamping fast remote servers.

4.2 Proximity, Locality and Availability

Each CoDNS node gathers and manages a set of neighbor nodes within a reasonable latency boundary. When a CoDNS instance starts, it sends a heartbeat to each node in the preconfigured CoDNS node list every second. The response contains the round trip time (RTT) and the average response time of the local nameserver at the peer node, reflecting the proximity and the availability of the peer node's nameserver. The top 10 nodes with different nameservers are picked as neighbors by comparing the sum with all nodes. Liveness of the chosen neighbors is periodically checked to see if the service is still available. One heartbeat is sent each second, so we guarantee the availability in 10 second granularity. Dead nodes are replaced with the next best node in the list.

Among these neighbor nodes, one peer is chosen for each remote name lookup using the Highest Random Weight (HRW) hashing scheme [22]. HRW consists of hashing the node name with the lookup name, and then choosing the node name with the smallest resulting hash value. Because HRW consistently picks the same node for the same domain name, this process enhances request locality for remote queries. Another desirable property of this approach is that some request locality is preserved as long as neighbor sets have some overlap. Full overlap is not required.

The number of neighbors is configurable according to the distribution of nodes. In the future, we will make CoDNS dynamically find the peer nodes not depending on the preconfigured set of nodes. One possible solution is to make each CoDNS node advertise its neighbor set and have a few well known nodes. Then, a new CoDNS node with no information about available CoDNS peer nodes can ask the well known nodes for their peer nodes and recursively gather the nodes by asking each neighbor until it finds a reasonable pool of CoDNS nodes.

Note that our neighbor discovery mechanisms are essentially advisory in nature – once the node has enough peers, it only needs to poll other nodes in order to have a reasonable set of candidates in case one of its existing peers becomes unavailable. In the event that some sites have enough peers to make this polling a scalability issue, each node can choose to poll a nearby subset of all possible peers to reduce the background traffic.

4.3 Policy & Tunability

In the future, we expect CoDNS node policy will become an interesting research area, given the tradeoffs between overhead and latency. We have made choices for initial delay and retry behavior for our environment, and we believe that these choices are generally reasonable. However, some systems may choose to tune CoDNS to have

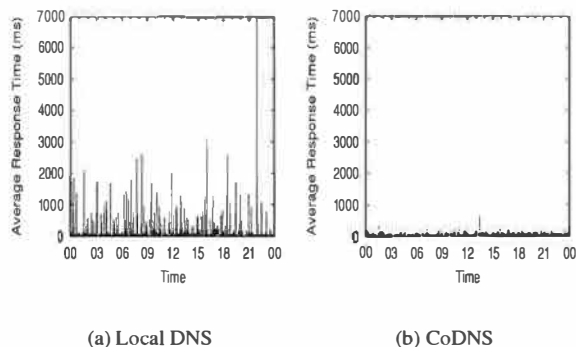


Figure 13: Minute-level Average Response Time for One Day on planetlab1.cs.cornell.edu

much lower overhead, at the cost of some latency benefit. In particular, systems that want to use it only to avoid situations where all local nameservers have failed could use an initial delay threshold of several seconds. In this case, if the local nameserver repeatedly fails to resolve requests in multiple seconds, the initial delay will drop to zero and all lookups will be handled remotely for the duration of the outage.

Sites may also choose to limit CoDNS overhead to a specific level, which would turn parameter choices into an optimization problem. For example, it may be reasonable to ask questions of the form “what is the best latency achievable with a maximum remote lookup rate of 10%?” Our trace-driven simulations give some insight into how to make these choices, but it may be desirable to have an online system automatically adjust parameter values continuously in order to meet these constraints. We are investigating policies for such scenarios.

4.4 Bootstrapping

CoDNS has a bootstrapping problem, since it must resolve peer names in order to operate. In particular, when the local DNS service is slow, resolving all peer names before starting will increase CoDNS’s start time. So, CoDNS begins operation immediately, and starts resolving peer names in the background, which greatly reduces its start time. The background resolver uses CoDNS itself, so as soon as a single working peer’s name is resolved, it can then quickly help resolve all other peer names. With this bootstrapping approach, CoDNS starts virtually instantaneously, and can resolve all 350 peer names in less than 10 seconds, even for slow local DNS. A special case of this problem is starting when local DNS is completely unavailable. In this case, CoDNS would be unable to resolve even any peer names, and could not send remote queries. CoDNS periodically stores all peer information on disk, and uses that information at startup. This file is shipped with CoDNS, allowing operation even on nodes that have no DNS support at all.

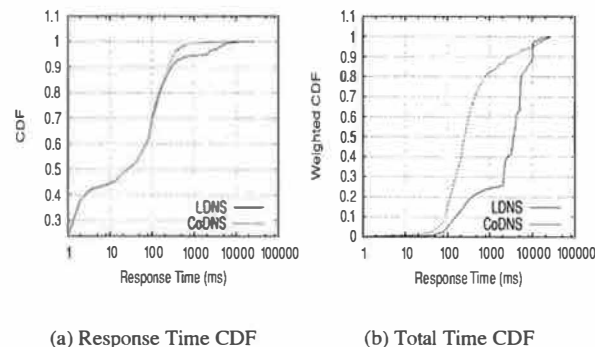


Figure 14: CDF and Weighted CDF for One Week on planetlab1.cs.cornell.edu, LDNS = local DNS

5 Evaluation / Live Traffic

To gauge the effectiveness of CoDNS, we compare its behavior with local DNS on CoDeeN’s live traffic using a variety of metrics. CoDeeN receives 5-7 million requests daily from a world-wide client population of 7-12K users. These users have explicitly specified CoDeeN proxies in their browser, so all of their Web traffic is directed through CoDeeN. The CoDeeN proxies maintain their own DNS caches, so only uncached DNS names cause lookups. To eliminate the possible caching effect on a nameserver from other users sharing the same server, we measure both times only in CoDNS, using the slaves to indicate local DNS performance.

CoDNS effectively removes the spikes in the response time, and provides more reliable and predictable service for name lookups. Figure 13 compares per-minute average response times of local DNS and CoDNS for CoDeeN’s live traffic for one day on one PlanetLab node. While local DNS shows response time spikes of 7 seconds, CoDNS never exceeds 0.6 seconds. The benefit stems from redirecting slow lookups to working peers.

The greater benefit of CoDNS lies in reducing the frequency of slow responses. Figure 14 shows a CDF and a weighted CDF for name lookup response distribution for the same node for one week. The CDF graph shows that the response distribution in both schemes is almost similar until the 90th percentile, but CoDNS reduces the lookups taking more than 1000ms from 5.5% to 0.6%. This reduction gives much benefit in total lookup time in the weighted CDF. It shows CoDNS now spends 18% of total time in lookups taking more than 1000ms, while local DNS still spends 75% of the total time on them.

This improvement is widespread – Figure 15(a) shows the statistics of 95 CoDeeN nodes for the same period. The average number of total lookups per node is 22,208, ranging from 12,119 to 131,466 per node. The average response time in CoDNS is 60-221ms, while that of local DNS is 113-935ms. In all cases, CoDNS’s response is

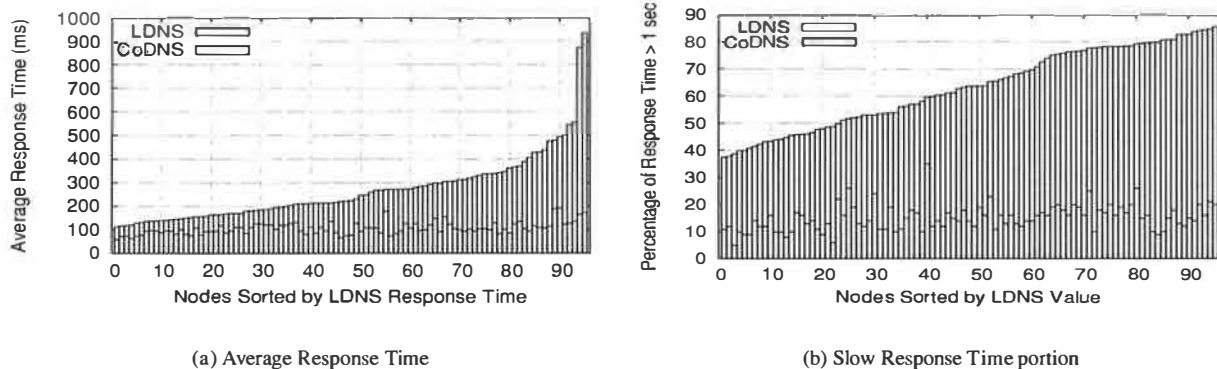


Figure 15: Live Traffic for One Week on the CoDeeN Nodes, LDNS = local DNS

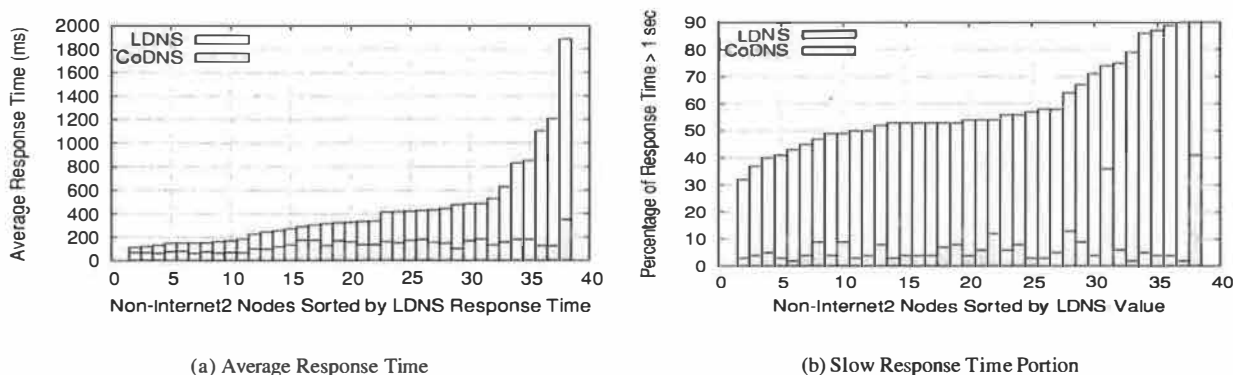


Figure 16: Non-Internet-2 Nodes, LDNS = local DNS

faster, ranging from a factor of 1.37 to 5.42. Figure 15(b) shows the percentage of slow responses in the total response time. CoDNS again reduces the slow response's portion dramatically to less than 20% of the total lookup time in most cases, delivering more predictable response time. In contrast, local DNS spends 37% to 85% of the total time in the slow queries.

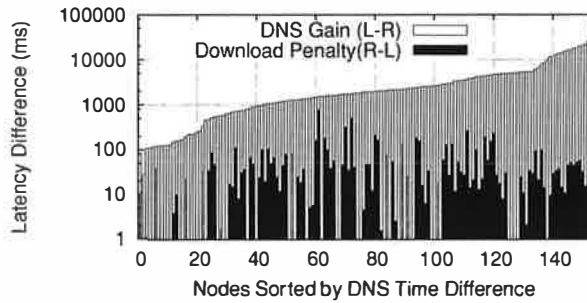
5.1 Non-Internet2 Benefits

Since most CoDeeN sites are hosted at North American universities with Internet2 (I2) connectivity, one may suspect that low-congestion I2 peer links are responsible for our benefits. To address this issue, we pick non-I2 PlanetLab nodes and replay 10,792 unique lookups of hostnames from one day's live traffic on a CoDeeN proxy. Figure 16(a) shows that CoDNS provides similar benefit on 38 non-I2 nodes as well. The average response time in CoDNS ranges from 63ms to 350ms, while local DNS is 113ms to 1884ms, an improvement of factor of 1.64 to 9.52. Figure 16(b) shows that CoDNS greatly reduces the slow response portion as well – CoDNS generally spends less than 10% of the total time in this range, while local DNS still spends 32% to 90%.

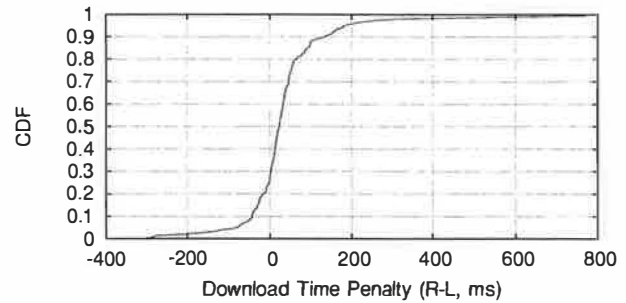
5.2 CDN Effect

CoDNS replaces slow local responses with fast remote responses, which may impact DNS-based CDNs [1] that resolve names based on which DNS nameserver sends the query. CoDNS may return the address of a far replica when it uses a peer's nameserver result. We investigate this issue by testing 14 popular CDN users including Apple, CNN, and the New York Times. We measure the DNS and download time of URLs for the logo image file on those web sites, and compare local DNS and CoDNS when their responses differ.

Since CoDNS is used only when the local DNS is slow or failing, it should come as no surprise that the total time for CDN content is still faster on CoDNS when they differ in returned IP address. The DNS time gain and the downloading time penalty presented in the difference between local and remote response time is shown in Figure 17(a). When local DNS is slow, CoDNS combined with a possibly sub-optimal CDN node is a much better choice, with the gain from faster name lookups dwarfing the small difference in download times when any difference exists. If we isolate the downloading time difference between the DNS-provided CDN node versus the



(a) DNS Lookup Time Gain vs. Downloading Time Penalty



(b) Cumulative Distribution of Downloading Time Difference

Figure 17: CDN Effect for www.apple.com, L = Local Response Time, R = Remote Response Time, DNS gain = Local DNS time - CoDNS time, Download penalty = download time of CoDNS-provided IP - download time of DNS-provided IP, shown in log scale. Negative penalties indicate CoDNS-provided IP is faster, and are not shown in the left graph.

CoDNS-provided CDN node, we get Figure 17(b). Surprisingly, almost a third of the CoDNS-provided nodes are closer than their DNS counterparts, and 83% of them show less than a 100ms difference. This matches the CDN's strategy to avoid notably bad servers instead of choosing the optimal server [8]. Results for other CDN vendors are similar.

5.3 Reliability and Availability

CoDNS dramatically improves DNS reliability, measured by the local nameserver availability. To quantify this effect, we measured the availability of name lookups for one month across all CoDeeN nodes, with and without CoDNS. We assume that a nameserver is available unless it fails to answer requests. If it fails, we consider the periods of time when no requests were answered as its unavailability. Each period is capped at a maximum of five seconds, and the total unavailability is measured as the sum of the unavailable periods. This data, shown in Figure 18, is presented using the reliability metric of "9's" of availability. Regular DNS achieves 99% availability on about 60% of the nodes, which means roughly 14 minutes per day of no service. In contrast, CoDNS is able to achieve over 99.9% availability on over 70% of nodes, reducing downtimes to less than 90 seconds per day. On some nodes, the availability approaches 99.99%, or roughly 9 seconds of unavailability per day. CoDNS provides roughly an additional '9' of availability, without any modifications to the local DNS infrastructure.

5.4 Overhead Analysis

To analyze CoDNS's overhead, we examine the remote query traffic generated by the CoDeeN live activity. For this workload, CoDNS issued 11% to 85% of the total lookups as remote queries, as shown in Figure 19. The variation reflects the health of the local nameserver, and less stable nameservers require more remote queries from CoDNS. Of the six nodes that had more than 50%

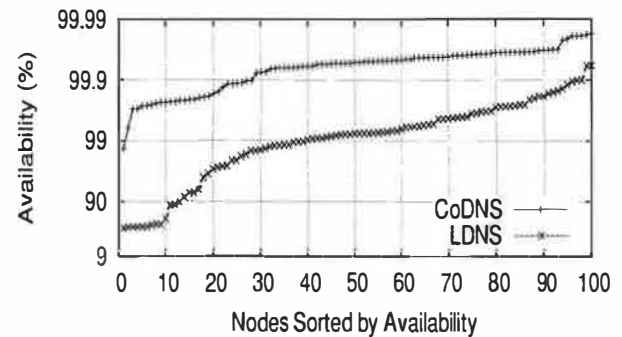


Figure 18: Availability of CoDNS and local DNS (LDNS)

remote queries, all experienced complete nameserver failure at some point, during which remote queries increased to over 100% of the local requests. These periods skew the average overhead.

We believe that the additional burden on nodes with working DNS is tolerable, due to the combination of our locality-conscious redirection and already high local nameserver hit rates. Using our observed median overhead of 25% and a local hit rate of 80% - 87% [9], the local DNS will incur only 3.25 - 5.00% extra outbound queries. Since remote queries are redirected only to lightly loaded nodes, we believe the extra lookups will be tolerable on the peer node's local nameserver.

We also note that many remote queries are not answered, with Figure 19 showing this number varies from 6% to 31%. These can be due to WAN packet losses, unresolvable names, and remote node rate-limiting. CoDNS nodes drop remote requests if too many are queued, which prevents a possible denial of service attack. CoDNS peers never reply if the request is unresolvable, since their own local DNS may be failing, and some other peer may be able to resolve the name.

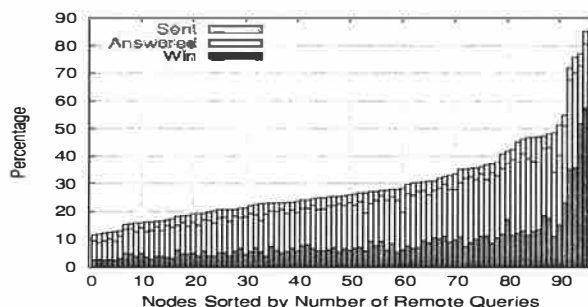


Figure 19: Analysis for Remote Lookups

The queries in which CoDNS “wins”, by beating the local DNS, constitute 2% to 57% of the total requests. On average, 9% of the original queries were answered by the remote responses, removing 47% of the slow response portion in the total lookup time shown in the Figure 15(b). Of the winning remote responses, more than 80% were answered by contacting the first peer, specified as “win-by-1” in Figure 20. Of all winning responses, 95% are resolved by the first or second peer, and only a small number require contacting three or more peers. This information can be used to further reduce CoDNS’s overhead by reducing the number of peers contacted – if it has not been resolved within the first three peers, then further attempts are unlikely to resolve it, and no more peers should be contacted. We may explore this optimization in the future, but our current overheads are low enough that we have no pressing need to reduce them.

In terms of extra network traffic generated for remote queries, each query contains about 300 bytes of a request and a response. On average, each CoDNS on a CoDeeN node handles 414 to 10,287 requests per day during the week period, amounting to 243KB to 6027KB. CoDNS also consumes heartbeat messages to monitor the peers each second, which contains 32 bytes of data. In sum, each CoDNS on a CoDeeN node consumes on average 7.5 MB of extra network traffic per day, consuming only 0.2% of total CoDeeN traffic in relative terms.

5.5 Application Benefits

By using CoDNS, CoDeeN obtains other benefits in capacity and availability, and these may apply to other applications as well. The capacity improvements come from CoDeeN being able to use nodes that are virtually unusable due to local DNS problems. At any given time, roughly 10 of the 100 PlanetLab nodes that run CoDeeN are experiencing significant DNS problems, ranging from high failure rates to complete failure of the primary (and even secondary) nameservers. CoDeeN nodes normally report their local status to each other, and before CoDNS, these nodes would tell other nodes to avoid them due to the DNS problems. With CoDNS,

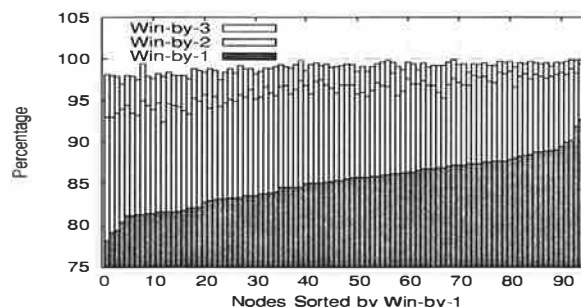


Figure 20: Win-by-N for Remote Lookups

these nodes can still be used, providing an additional 10% extra capacity.

The availability improvements come from reducing startup time, which can be dramatic on some nodes. CoDeeN software upgrades are not announced downtimes, because on nodes with working local DNS, CoDeeN normally starts in 10-15 seconds. This startup process is fast enough that few people notice a service disruption. Part of this time is spent in resolving the names of all CoDeeN peers, and when the primary DNS server is failing, each lookup normally requires over five seconds. For 120 peers, this raises the startup time to over 10 minutes, which is a noticeable service outage. If CoDNS is already running on the node, startup times are virtually unaffected by local failure, since CoDNS is already sending all queries to remote servers in this environment. If CoDNS starts concurrently with CoDeeN, the startup time for CoDeeN is roughly 20 seconds.

6 Other Approaches

6.1 Private Nameservers

Since local nameservers exhibit overload, one may be tempted to run a private nameserver on each machine, and have it contact the global DNS hierarchy directly. This approach is more feasible as a backup mechanism than as a primary nameserver for several reasons. Using shared nameservers reduces maintenance issues, and the shared cache can be larger than individual caches. Not only does cache effectiveness increase due to capacity, but the compulsory misses will also be reduced from the sharing. With increased cache misses, the global DNS failure rate becomes more of an issue, so using private nameservers may reduce performance and reliability.

As a backup mechanism, this approach is possible, but has the drawbacks common to any infrequently-used system. If it is not being exercised regularly, failure is less likely to be noticed, and the system may be unavailable when it is needed most. It also consumes resources when not in use, so other tasks on the same machine will be impacted, if only slightly.

6.2 Secondary Nameservers

Since most sites have two or more local nameservers, another approach would be to modify the resolver libraries to be more aggressive about using multiple nameservers. Possible options include sending requests to all nameservers simultaneously, being more aggressive about timeouts and using the secondary nameserver, or choosing whichever one has better response times.

While we believe that some of these approaches have some merit, we also note that they cannot address all of the failure modes that CoDNS can handle. In particular, we have often seen all nameservers at a site fail, in which case CoDNS is still able to answer queries via the remote nameservers. Correlated failure of local nameservers renders these approaches useless, while correlated failure among groups of remote servers is less likely.

Overly aggressive strategies are likely to backfire in the case of local nameservers, since we have seen that overload causes local nameserver failure. Increasing the request rate to a failing server is not likely to improve performance. Load balancing among local nameservers is more plausible, but still requires modifications to all clients and programs. Given the cost of changing infrastructure, it is perhaps appealing to adopt a technique like CoDNS that covers a broader range of failures.

Finally, upgrade cost and effort are real issues we have heard from many system administrators – secondary nameservers tend to be machines that are a generation behind the primary nameservers, based on the expectation of lower load. Increasing the request rate to the secondary nameserver will require upgrading that machine, whereas CoDNS works with existing infrastructure.

6.3 TCP Queries

Another possible solution is to use TCP instead of UDP as a way of communicating with local nameservers. If the failure is caused by packet losses in the LAN or silent packet drops caused by UDP buffer overflow, TCP can improve the situation by reliable data delivery. In addition, the flow control mechanism inherent in TCP can ask the name lookup clients to slow down when the nameserver is overloaded.

Although the DNS RFC [14] allows the use of TCP in addition to UDP, in practice, TCP is used only when handling AXFR queries for the zone transfer or when the requested record set is bigger than 512 bytes. The reason why TCP is not favored in name lookups is mainly because of the additional overhead. If a TCP connection is needed for every query, it would end up handling nine packets instead of two: three to establish the connection, two for the request/response, and four to tear down the connection. A persistent TCP connection might remove the per-query connection overhead, but it also needs to consume one or two extra network packets for ACKs.

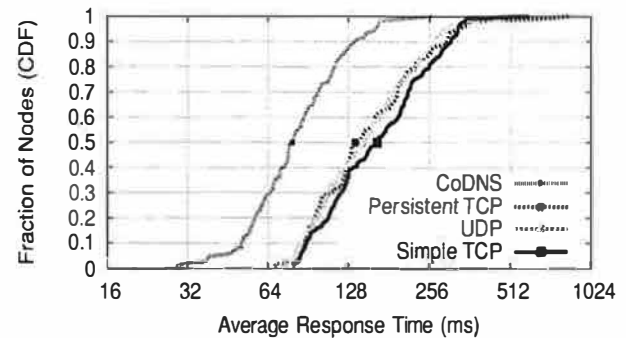


Figure 21: Comparison of UDP, TCP, and CoDNS latencies

Also, there is another issue of reclaiming the idle connections, since they consume system resources and can degrade performance. The DNS RFC [14] specifies two minutes as a cutoff but in practice most servers disconnect the idle connection within 30 seconds.

To compare the performance between UDP and TCP, we replay 10,792 unique hostnames obtained from one day's live traffic of a CoDeeN proxy at 107 PlanetLab nodes. Carrying out a completely fair comparison is difficult, since we cannot issue the same query for all of them at the same time. Instead, to give a relatively fair comparison, we run the test for CoDNS first, and subsequently run other parts, making all but CoDNS get the benefit of cached responses from the local nameserver after having been fetched by CoDNS. Figure 21 shows the CDF of the average response time for all approaches. Persistent TCP and UDP have comparable performance, while simple TCP is noticeably worse. The CoDNS latencies, included for reference, are better than all three.

The replay scenario described above should be favorable to TCP, but even in this conservative configuration, CoDNS still wins. Figure 22(a) shows that all nodes report that CoDNS is 10% to 500% faster than TCP, confirming CoDNS is a more attractive option than TCP. The large difference is in the slow-response portion, where CoDNS wins the most and where TCP-based lookups cannot help. Figure 22(b) shows that a considerable amount of time is still spent on the long delayed queries in TCP-based lookups. CoDNS reduces this time by 16% to 92% when compared to the TCP-based measurement. Though TCP ensures that the client's request reaches the nameserver, if the nameserver is overloaded, it may have trouble contacting the DNS hierarchy for cache misses.

7 Related Work

Traditional DNS-related research has focused on the problems in the server-side DNS infrastructure. As a seminal study in DNS measurement, Danzig *et al.* found that a large number of network packets were being wasted due to DNS traffic, blaming nameserver software bugs and misconfigurations as major culprits [5].

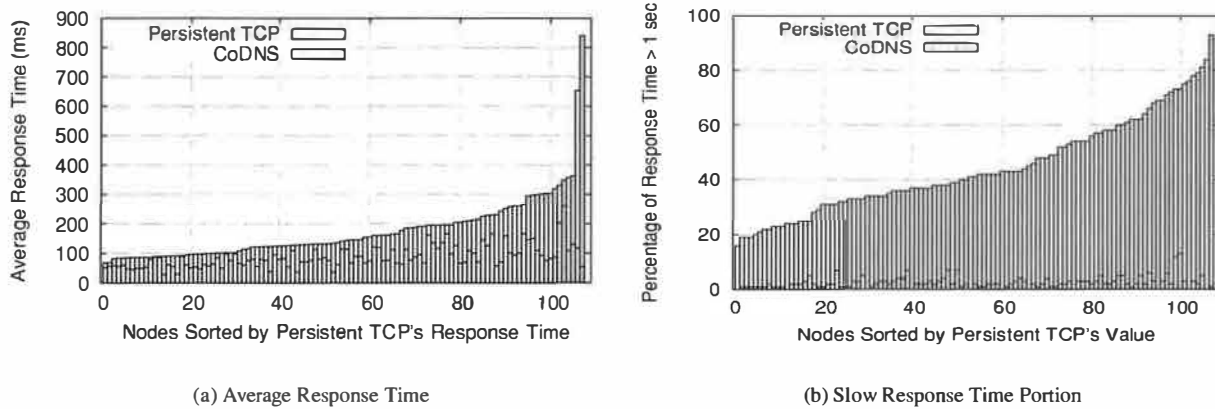


Figure 22: CoDNS vs. TCP

Since then, bugs in the resolvers and nameservers have been reduced [12], but recent measurements show that there is still much room for improvement. In 2000, Wills *et al.* [24] and Huitema *et al.* [7] reported 29% of DNS lookups take over 2 seconds, and Cohen *et al.* [3] reported 10% of lookups exceed more than 3 seconds. Jung *et al.* also present data indicating 23% of all server-side lookups receive no results, indicating the problems of improper configurations and incorrect nameservers still persist [9]. They measure the client-side performance in terms of response time and caching hit ratio as well. However, that work does not trace the origins of name lookup delays from the client-side, concentrating only on the wide-area DNS traffic. Given the fact that local nameserver cache hit ratios are 80% - 87% [9, 24], even a small problem in the local nameserver and its environment can skew the latency of a large number of lookups. Our study addresses this problem. Liston *et al.* indirectly provide some evidence of local nameserver problems by attributing the major sources of response time delay to end nameservers rather than the root/gTLD servers [13].

The research community has recently renewed its focus on improving server-side infrastructure. Cox *et al.* investigate the possibility of transforming DNS into a peer-to-peer system [4] using a distributed hash table [21]. The idea is to replace the hierarchical DNS name resolving process with a flat peer-to-peer query style, in pursuit of load balancing and robustness. With this design, the misconfigurations from mistakes by administrators can be eliminated and the traffic bottleneck on the root servers are removed so that the load is distributed over the entities joining the system.

In CoDoNS, Ramasubramanian *et al.* improve the poor latency performance of this approach by using proactive replication of DNS records [18]. They exploit the Zipf-like distribution of the domain names in web browsing [2] to reduce the replicating overhead while

providing $O(1)$ proximity [17]. Our approaches differ in several important aspects – we attempt to reduce overlapping information in caches, in order to maximize the overall aggregate cache size, while they use replication to reduce latency. Our desire for a small process footprint stems from our observation that memory pressure is one of the causes of current failures in client-side infrastructure. While their system appears not to be deployed in production, they perform an evaluation using a DNS trace with a Zipf factor above 0.9 [18]. In comparison, our evaluation of CoDNS uses the live traffic generated by CoDeeN *after* its proxies have used their local DNS caches, so the request stream seen by CoDNS has a Zipf factor of 0.50-0.55, which is a more difficult workload. We intend to compare the live performance of CoDNS versus CoDoNS when the latter system enters production and is made available to the public. In any case, since CoDNS does not depend on the specifics of the name lookup system, we expect that it can interoperate with CoDoNS if the latter provides better performance than the existing nameservers at PlanetLab sites. One issue that will have to be addressed by any proposed DNS replacement system is the use of intelligent nameservers that dynamically determine which IP address to return for a given name. These nameservers are used in CDNs and geographic load balancers, and can not be replaced with purely static lookups, such as those performed in CoDoNS. Since CoDNS does not replace existing DNS infrastructure, we can interoperate with these intelligent nameservers without any problem.

Kangasharju *et al.* pursue a similar approach to reducing the DNS lookup latency by more aggressively replicating DNS information [10]. Inspired by the fact the entire DNS record database fits into the size of a typical hard disk and with the recent emergence of terrestrial multicast and satellite broadcast systems, this scheme reduces the need to query the distant nameservers by keep-

ing the DNS information up to date by efficient world-wide replication.

The difference in our approach is to temporarily use functioning nameservers of peer nodes, separate from the issue of improving the DNS infrastructure itself. We expect that benefits in improving the infrastructure “from above” will complement our “bottom up” approach. One advantage of our system is that misconfigurations can be masked without name server outage, allowing administrators more time to investigate the problem.

8 Conclusion

We have shown that client-side instability in DNS name lookups is widespread and relatively common. The failure cases degrade average lookup time and increase the “tail” of response times. We show that these failures appear to be caused by temporary nameserver overload, and are largely uncorrelated across multiple sites. Through analysis of live traffic, we show that a simple peering system reduces response times and improves reliability.

Using these observations, we develop a lightweight name lookup service, CoDNS, that uses peers at remote sites to provide cooperative lookups during failures. CoDNS operates in conjunction with local DNS nameservers, allowing incrementally deployment without significant resource consumption. We show that this system generates low overhead, cuts average response time by half or more, and increases DNS service availability.

Acknowledgments

This research was supported in part by NSF grant CNS-0335214. We would like to thank Jeffrey Mogul and Hewlett-Packard for donation of the Alpha workstation used for testing. We thank our shepherd, Geoff Voelker, for his guidance and helpful feedback, and we thank our anonymous reviewers for their valuable comments on improving this paper.

References

- [1] Akamai. Content Delivery Network. <http://www.akamai.com>.
- [2] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of IEEE INFOCOM*, 1999.
- [3] E. Cohen and H. Kaplan. Prefetching the Means for Document Transfer: A New Approach for Reducing Web Latency. In *Proceedings of IEEE INFOCOM*, 2000.
- [4] R. Cox, A. Muthithachoen, and R. Morris. Serving DNS Using Chord. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.
- [5] P. B. Danzig, K. Obraczka, and A. Kumar. An Analysis of Wide-Area Name Server Traffic: A Study of Internet Domain Name System. In *Proceedings of ACM SIGCOMM*, 1992.
- [6] D. Eastlake. Domain Name System Security Extensions. RFC 2535, January 1999.
- [7] C. Huitema and S. Weerahandi. Internet Measurements: the Rising Tide and the DNS Snag. In *Proceedings of the 13th ITC Specialist Seminar on Internet Traffic Measurement and Modelling*, 2000.
- [8] K. Johnson, J. Carr, M. Day, and F. Kaashoek. The Measured Performance of Content Distribution Networks. In *Proceedings of the 5th International Web Caching and Content Delivery Workshop (WCW)*, 2000.
- [9] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. DNS Performance and the Effectiveness of Caching. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop*, 2001.
- [10] J. Kangasharju and K. W. Ross. A Replicated Architecture for the Domain Name System. In *Proceedings of IEEE INFOCOM*, 2000.
- [11] B. Knowles. Domain Name Server Comparison: BIND 8 vs. BIND 9 vs. djbdns vs. ???, 2002. http://www.usenix.org/events/lisa02/tech/presentations/knowles_ppt/.
- [12] A. Kumar, J. Postel, C. Neuman, P. Danzig, and S. Miller. Common DNS Implementation Errors and Suggested Fixes. RFC 1536, October 1993.
- [13] R. Liston, S. Srinivasan, and E. Zegura. Diversity in DNS Performance Measures. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop*, 2002.
- [14] P. Mockapetris. Domain Names - Implementation and Specification. RFC 1035, November 1987.
- [15] P. Mockapetris and K. Dunlap. Development of the Domain Name System. In *Proceedings of ACM SIGCOMM*, 1988.
- [16] PlanetLab. <http://www.planet-lab.org>.
- [17] V. Ramasubramanian and E. G. Sirer. Beehive: O(1) Lookup Performance for Power-Law Query Distributions in Peer-to-Peer Overlays. In *1st Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.
- [18] V. Ramasubramanian and E. G. Sirer. The Design and Implementation of a Next Generation Name Service for the Internet. In *Proceedings of ACM SIGCOMM*, 2004.
- [19] M. Schiffman. A Sampling of the Security Posture of the Internet's DNS Servers. <http://www.packetfactory.net/papers/DNS-posture/>.
- [20] A. Shaikh, R. Tewari, and M. Agrawal. On the Effectiveness of DNS-based Server Selection. In *Proceedings of IEEE INFOCOM*, 2001.
- [21] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of ACM SIGCOMM*, San Diego, California, 2001.
- [22] D. Thaler and C. Ravishanker. Using Name-based Mappings to Increase Hit Rates. In *IEEE/ACM Transactions on Networking*, volume 6, 1, 1998.
- [23] L. Wang, K. Park, R. Pang, V. Pai, and L. Peterson. Reliability and Security in the CoDeeN Content Distribution Network. In *USENIX Annual Technical Conference*, 2004.
- [24] C. E. Wills and H. Shang. The Contribution of DNS Lookup Costs to Web Object Retrieval. Technical Report WPI-CS-TR-00-12, Worcester Polytechnic Institute (WPI), 2000.

Middleboxes No Longer Considered Harmful

Michael Walfish, Jeremy Stribling, Maxwell Krohn,
Hari Balakrishnan, Robert Morris, and Scott Shenker*
MIT Computer Science and Artificial Intelligence Laboratory
<http://nms.csail.mit.edu/doa>

Abstract

Intermediate network elements, such as network address translators (NATs), firewalls, and transparent caches are now commonplace. The usual reaction in the network architecture community to these so-called middleboxes is a combination of scorn (because they violate important architectural principles) and dismay (because these violations make the Internet less flexible). While we acknowledge these concerns, we also recognize that middleboxes have become an Internet fact of life for important reasons. To retain their functions while eliminating their dangerous side-effects, we propose an extension to the Internet architecture, called the *Delegation-Oriented Architecture* (DOA), that not only allows, but also facilitates, the deployment of middleboxes. DOA involves two relatively modest changes to the current architecture: (a) a set of references that are carried in packets and serve as persistent host identifiers and (b) a way to resolve these references to delegates chosen by the referenced host.

1 Introduction

The Internet's architecture is defined not just by a set of protocol specifications but also by a collection of general design guidelines. Among the architecture's original principles [12] are two tenets at the network layer (*i.e.*, IP layer) that are still widely valued, but are nonetheless often disobeyed in the current Internet:

#1: Every Internet entity has a unique network-layer identifier that allows others to reach it. During the Internet's youth, every network entity had a globally unique, fixed IP address. However, the emergence of private networks and host mobility, among other things, ended the halcyon days of unique identity and transparent reachability. Now, many Internet hosts have no globally unique handle that serves to direct packets to them.

#2: Network elements should not process packets that are not addressed to them. We call this tenet "network-level layering", and it implies that only a network element identified by an IP packet's destination field should inspect the packet's higher-layer fields.

This decades-old guideline has become an empty commandment, as firewalls, network address translators (NATs), transparent caches, and other widely deployed network elements use higher-layer fields to perform their functions.

That these tenets are routinely violated is not merely an Internet legalism. The inability of hosts in private address realms to pass handles allowing other hosts to communicate with them has hindered or halted the spread of newer protocols, such as SIP [24] and various peer-to-peer systems [18]. Layer violations lead to rigidity in the network infrastructure, as the transgressing network elements may not accommodate new traffic classes. The hundreds of IETF proposals for working around problems introduced by NATs [54], firewalls, and other layer-violating boxes are compelling evidence that *middleboxes* (as such hosts are collectively known) and the Internet architecture are not in harmony [8]. Indeed, because middleboxes violate one or both tenets above, Internet architects have traditionally reacted to them with disdain and despair.

We take a different view. Rather than seeing middleboxes as a blight on the Internet architecture, we see the current Internet architecture as an impediment to middleboxes. We believe such *intermediaries*, as we will call them, exist for important and permanent reasons, and we think the future will have more, not fewer, of them.

The market will continue to demand intermediaries for various reasons. NATs maintain and bridge between different IP spaces.¹ Firewalls and other boxes that intercept unwanted packets will be increasingly needed as attacks on end-hosts grow in rate and severity. Since even sophisticated users have difficulty configuring PCs to be impervious to attack, we believe that users would want to outsource this protection to a professionally managed host—one not physically interposed in front of the user—that would vet incoming packets. Under the current architecture, such outsourcing to "off-path" hosts requires special-purpose machinery and extensive manual configuration. Intermediaries can also increase

¹Even if the move to IPv6 accelerates, some IPv4 networks will remain. Moreover, private address realms give some protection against certain types of network attacks. Hence, we do not think private IP spaces are a temporary inconvenience that will soon end.

*Affiliation: UC Berkeley and ICSI.

performance through, for example, caching and load-balancing. Commercial service providers will continue to take advantage of such performance-enhancing intermediaries, disregarding architectural purity.

Thus, we have a fundamental conflict: although intermediaries offer clear advantages, they run afoul of the two tenets above, which causes harm and makes deploying and using intermediaries unnecessarily hard. Our goal, therefore, is an architecture hospitable to intermediaries, specifically one that allows intermediaries to abide by the two tenets, to avoid other architectural infractions, and to retain the same functions as today. Such an architecture would let a variety of middleboxes be deployed while also letting end-system protocols evolve independently and quickly.

Our architecture—which we call the *Delegation-Oriented Architecture* (DOA)—is based on two main ideas. First, all entities have a *globally unique identifier in a flat namespace*, and packets carry these identifiers. Second, DOA allows senders and receivers to express that one or more intermediaries should process packets en route to a destination. This *delegation* lets the resulting architecture embrace intermediaries as first-class citizens that are explicitly invoked and need not be physically interposed in front of the hosts they service. Globally unique identifiers and delegation have existed in previous work describing different architectures (e.g., i3 [57]); see §9 for details. This paper’s contribution is defining a relatively incremental extension to the Internet architecture, DOA, that coherently accommodates network-level intermediaries like NATs and firewalls. DOA requires no changes to IP or IP routers but does require changes to host and intermediary software. However, these changes are modular, and current applications can be easily ported.

We illustrate DOA with two examples: first, “network-extension boxes” which are analogous to today’s NATs in their establishment of private addressing realms but do not obscure hosts’ identities, and second, “network filtering boxes” which are analogous to today’s firewalls but do not violate network-level layering and need not be topologically in front of the hosts they service. Our goal is to show how our architecture easily accommodates these boxes.

Scope and Limitations

DOA is based on a subset of the architecture in a previous paper [3]. That position paper touches on various issues—including mobility, multi-homing, network-level middleboxes, and application-level middleboxes—with scant attention to design details or implementations. In an attempt to bring some of those nebulous architectural mutterings into focus, this paper concentrates exclusively on network-level intermediaries and ignores their

application-level counterparts.² This paper does not discuss mobility and multi-homing scenarios either (though DOA, because it separates location and identity, could—with modest extensions—handle those scenarios). Given our limited focus, DOA should be viewed not as a comprehensive architecture but rather as an architectural component designed to address network-layer middleboxes. Its design presumes IPv4 at the network layer but DOA is also compatible with, and useful for, IPv6.

The final limitation we mention is that some people want to deploy tenet-violating middleboxes (e.g., a censorious government that silently filters packets entering and exiting national borders) and that DOA can neither prevent such architecturally suspect middleboxes nor mitigate their damage.

2 Background

This review of common network-layer middleboxes is limited to the two we build under DOA—NATs and firewalls—and to a subset of their drawbacks; for a complete review, see [8, 18, 23, 38, 55]. Although NAT and firewalling are often combined in one box, these two functions are logically separate.

2.1 NAT and NAPT

Network Address Translation (NAT) and Network Address Port Translation (NAPT) [54] have several uses. For the purposes of this paper, we assume the following common scenario: a NAT or NAPT box bridges two address realms, at least one of which is *private*. Private addresses are unique within an address realm but ambiguous between address realms [46]; public addresses are globally unique and reachable from all Internet hosts. The hosts in the private realm are said to be *behind* the box. Packets destined for hosts behind the box are said to be *inbound*. The difference between NAT and NAPT is that NATs do not look at fields beyond the IP header. We adopt the convention of referring to both NAT and NAPT as “NAT”, though our description focuses on NAPT (the more common of the two today); for simplicity, we assume that NAPT has only one external IP address.

People deploy NATs for two reasons:

- *Convenience and Flexibility*: Private addressing realms allow people to administer a set of hosts without having to obtain public IP addresses for each.
- *Security*: Since hosts behind the NAT do not have global identities, a host outside the private realm cannot address the hosts in the private realm or express that traffic should go to them, which protects them from unwanted traffic. Also, by default (*i.e.*, without manual configuration), a NAT allows only inbound

²The basic architectural ideas can be illustrated with network-level intermediaries. At the application level, one must consider how applications are structured and named, a topic outside this paper’s scope [3].

traffic that is part of a connection initiated by a host behind the NAT.

The main operations performed by a NAT are: (1) dynamically allocating a source port at its public IP address when a host behind it initiates a TCP connection or sends a UDP packet; and (2) rewriting IP address and transport-layer port fields to demultiplex inbound packets to the hosts behind the NAT and to multiplex outbound packets over the same source IP address. NATs violate both tenets in §1. First, a NATed host's conception of its identity (namely its IP address) is a private address and thus is not a handle that it can pass around to allow other network entities to reach it. Second, NATs' modification of port fields violates tenet #2.

NATs cause the following additional problems:

- In order for a server behind a NAT to receive unsolicited inbound packets sent to a given destination port, one must statically configure the NAT with instructions about packets with that destination port. This manual step is called *hole punching* and requires administrative control over the NAT. The amount of manual configuration increases when a series of NATs separate a server from the public Internet, creating a tree of private address spaces³—in this case, one must not only configure each of the NATs in the tree but also coordinate among them; *e.g.*, each globally reachable Web server in a given tree of NATs must get traffic on a different port on the outermost NAT's public IP address. (By *outermost*, we mean “connected to the globally reachable Internet”.)
- Hosts behind the same NAT cannot simultaneously receive traffic sent to the same TCP port number on the NAT's public IP address. However, some applications require traffic on a specific port; *e.g.*, IPSEC expects traffic on port 500 [44], so only one host within a tree of NATs can receive Virtual Private Network (VPN) [21] service.

2.2 Firewalls

A firewall blocks certain traffic classes on behalf of a host by examining IP-, transport-, and sometimes application-level fields and then applying a set of “firewall rules”. It must be on the topological path between the host and the host's Internet provider, which we argue is unnecessarily restrictive. Today's firewalls disobey tenet #2 because, by design, they must inspect many non-IP fields in each packet. Since firewalls by default distrust that which they do not recognize, they may block novel but benign traffic, even if the intended recipient wants to allow the traffic.

³Such series of NATs are not artificial; see §5.4 and Figure 4.

3 Architectural Overview of DOA

This section gives an overview of DOA; we defer design details to §4. We first list desired architectural properties that aid in gracefully accommodating intermediaries and then describe mechanisms to achieve those properties. The remainder of the section discusses how DOA extends the Internet architecture.

3.1 Desired Architectural Properties

(1) Global identifiers in packets: Each packet should contain an identifier that unambiguously specifies the ultimate destination. The Internet architecture, as originally conceived, *did* provide global identifiers in packets, but IPv4 addresses no longer meet the “global identifier” requirement. (IPv6 addresses, because they reflect network topology, are also unsuitable for us, as we elaborate below.) The purpose of a global identifier is to uniquely identify the packet's ultimate destination to intermediaries in a way that is application-independent.

(2) Delegation as a primitive: Hosts should have an application-independent way to express to others that, to reach the host, packets should be sent to an intermediary or series of intermediaries. This primitive—called *delegation*—allows end-hosts or their administrators to explicitly invoke (and revoke) intermediaries. These intermediaries need not be “on the topological path”.

3.2 Mechanisms

EIDs: To achieve property (1), each host has an unambiguous endpoint identifier picked from a large namespace. Our design imposes the following additional requirements:

- (a) The identifier is independent of network topology (ruling out IPv6 addresses and other identifiers with topology-dependent components, as in [42, 43]). With this requirement, hosts can change locations while keeping the same identifiers.
- (b) The identifier can carry cryptographic meaning (ruling out human-friendly DNS names). We explain the purpose of this requirement later in this section.

To satisfy these requirements, we choose flat 160-bit endpoint identifiers (EIDs). A DOA header between the IP and TCP headers carries source and destination EIDs. Transport connections are bound to source and destination EIDs (instead of to source and destination IP addresses as in the status quo). DOA borrows the idea of topology-independent EIDs from previous work, including Nimrod [34], HIP [39], UIP [17].

EIDs are resolved . . . : DOA provides for delegation as a primitive by *resolving* EIDs. We presume a mapping service, accessible to Internet hosts, that maps

EIDs to some target specified by the EID owner. This resolution has two flavors:

- **... to IP addresses:** In order to communicate with an end-host identified by an EID, a prospective peer uses the mapping service to resolve the EID to an IP address. This indirection creates a way for a host to specify that prospective peers should direct their packets to a given delegate: the host has its EID resolve to the IP address of the delegate.
- **... to other EIDs:** More generally, an EID can resolve to another EID, allowing an end-host to map its EID to a delegate's *identity*; if an end-host's EID had to map to the delegate's *IP address* (or any other topology-dependent identifier), the end-host would have to update the mapping whenever the delegate's location changed. An EID can also resolve to a *sequence* of EIDs, each of which identifies an intermediary specified by the host. This sequence is carried in packets, yielding a loose *source route* in identifier space.⁴ This option is reminiscent of i3's stacked identifiers.

Thus, our design requires an EID resolution infrastructure. We wish the management of this infrastructure to be as automated as possible, which is why we had requirement (b), above: automated management is easier if the EIDs are vested with cryptographic meaning [36]. The resolution infrastructure must scalably support a put()/get() interface over a large, sparse, and flat namespace. Distributed hash tables (DHTs) [2, 14, 49, 62] give exactly this capability, but any other technology that offers this capability would also suffice. DNS's "resolve-your-own-namespace" economic model cannot be used here, but there are plausible scenarios for the economic viability of a DHT-based resolution infrastructure [61].

We have not yet mentioned *sender-invoked* intermediaries. Under DOA, senders invoke intermediaries by putting into packets additional identifiers beyond the identifiers that resulted from resolving the receiver's EID. Sender-invoked intermediaries receive little attention in this paper but are part of DOA's design.

3.3 DOA and the Two Tenets

We elaborate on our earlier claim that DOA allows intermediaries to abide by the two tenets in §1. Because they are location-independent and drawn from a massive namespace, EIDs can globally and unambiguously identify hosts, satisfying tenet #1. As a result, a network element can reply to the source of a packet by sending to the location given by the resolution of the source EID.

To obey network-level layering (tenet #2), network elements need only follow normal IP layering rules, as follows. If an IP packet arrives at a network element

and the packet's destination IP address is not the network element's, then the element may change nothing in the packet besides per-hop fields. (However, elements may drop packets based on information in the IP header, which permits functions such as ingress and egress filtering.) If, on the other hand, the packet's destination IP address matches the network element's, there are two cases: (1) The destination EID in the DOA header matches the network element's EID (*i.e.*, the packet has reached its destination); or (2) These EIDs do not match, which means the element is a delegate. In the latter case, network-level layering implies that the allowed packet operations are up to the entities in the delegation relationship.

Note that this last claim satisfies network-level layering but allows violations of higher-level equivalents, *e.g.*, an explicitly addressed firewall that looks at application payloads upholds the rules just given but flouts application-level layering. In general, this paper claims that DOA improves on the status quo by restoring network-level layering but does not insist that intermediaries adhere to higher-level layering. Why not? Higher-level layers define how to organize host software, and one can imagine splitting host software among boxes using exotic decompositions. Defining both higher-level layering and an architecture that respects these higher layers is a problem that requires care and one we have left to future work. In the meantime, we believe that hosts invoking intermediaries should decide how best to split functions between them and their intermediaries.

We now discuss how the IP layering rules given above apply to specific intermediaries. Under DOA, NATs, which exist to bridge address realms, need not obscure host identity: as we describe in more detail in §5, DOA-based NATs may rewrite IP fields but will neither touch DOA fields that carry host identities nor overload transport-layer fields. Also, firewalls could be explicitly invoked, meaning that packets ending up at the firewalls would be addressed to the firewall. While these new firewalls (which we cover in §6) could certainly have outmoded policies, causing them to drop novel traffic classes just as today's firewalls do, they are not violating network-level layering because packets are addressed to them. One result of this explicit addressing is that the firewall's invocation is under users' (or their administrators') control, so the user (or administrator) could decide to have packets destined for it sent to *another* firewall, one with better suited policies.

3.4 DOA and Internet Evolvability

The preceding point is more general than firewalls and is important for the Internet's flexibility and evolvability. Today, there is only one easy way to deploy a middlebox: putting it "on the path". Of course, under DOA, some

⁴In this case, transport connections are bound to the ultimate endpoint, which is identified by the last EID in the sequence.

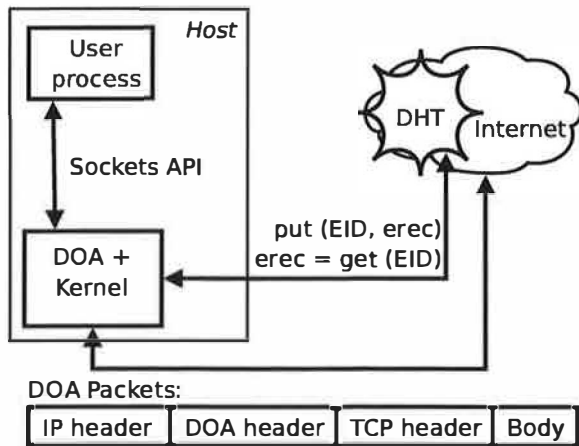


Figure 1: High-level view of DOA design.

boxes would have to be on the topological path to enforce physical security (e.g., for denial-of-service protection); §6.4 describes how DOA accommodates these on-path boxes. However, DOA—with its flexible and application-independent invocation primitive—also gives users or their administrators the option to *outsource* functionality. Thus, under DOA, fewer intermediaries would need to be physically interposed, and users, no longer limited to the capabilities of the boxes in front of them, could avail themselves of a menu of services.

As a result, we believe that DOA could permit the rise of a competitive market in professionally managed intermediary services such as firewalls. Delegation and resolution are precisely what is necessary for such a market—the ability for users to select a provider and to switch providers. Because users would have a choice, they could seek the intermediary service that best suited their needs, and because these services would be professionally managed, they could keep up with the rapid pace of application innovation. Thus, we see DOA as contributing to the Internet’s ability to evolve.

While we believe in its benefits, it is not clear that DOA is necessary for these new functions. In fact, we conjecture that even for those applications and intermediaries that one can seemingly build only under DOA, someone with enough imagination and fortitude could achieve equivalent functionality under the status quo—but not without running afoul of a basic tenet of the Internet architecture. We do suspect that the mechanisms of DOA will help new Internet functionality to evolve, but ultimately we believe our contribution is not novel *function* but rather novel *architecture*—making a class of network intermediary functions easier to build and reason about, and less likely to cause harm.

A natural question is how DOA relates to the canonical end-to-end argument [51], which is often interpreted as a warning against intermediaries. The central claim of the end-to-end argument is that application intelligence

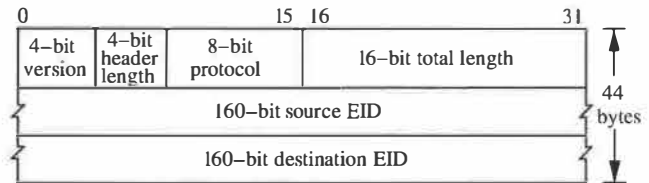


Figure 2: Example DOA header with no stacked identifiers.

is best implemented on end-hosts and not “in the network” because intelligence in the network leads to inflexibility and because end-hosts know best what functions they need. At a high level, DOA upholds this vision: the explicit invocation of intermediaries means that intelligence is not stuck in the network and that end-hosts can invoke the intermediaries that best serve them.

4 Detailed DOA Design

Given the preceding general description of DOA, we now present details of the design. Figure 1 shows the DOA components and the interfaces between them.

4.1 Header Format

DOA packets are delivered over IP, with the IP protocol field set to a well-known value. An example DOA header, with no extensions, is shown in Figure 2; the header length is measured in four-byte words, the protocol field is the transport-level protocol (e.g., TCP, UDP) used by the packet, and the length field gives the DOA packet’s total length (including the DOA header but not IP header) in bytes. TCP and UDP pseudo-checksums include the EIDs where IP addresses are used today (since transport logically occurs between two entities, each identified by an EID). The DOA header is extensible (e.g., the remote packet filter presented in §6 extends the basic DOA header).

4.2 Resolution and Invoking Intermediaries

A DOA host wishing to send a packet to a recipient obtains the recipient’s EID e out-of-band (e.g., by resolving the recipient’s DNS name to e). The sender then uses the EID resolution infrastructure—which is discussed in §3.2 and which we base on DHTs—to retrieve an *erecord*, depicted in Figure 3. An *erecord*’s fields are as follows: the EID field is the EID being resolved; the Target field is described in the next paragraph; the Hint field is optional information whose use we illustrate in §5; and the TTL field, like DNS’s TTL, is a hint indicating how long entities should cache the *erecord*. DOA presumes that EID owners (or administrators acting on their behalf) maintain and possibly periodically refresh⁵ the DHT’s copy of their *erecord*.

⁵Some DHTs, like OpenDHT [29], store only soft state, requiring EID owners to do refreshes.

EID:	0x345ba4d...
Target:	EID ⁺ or IP address
Hint:	e.g., IP address
TTL:	time-to-live, a caching hint

Figure 3: The **erecord**.

Recall from §3.2 that EIDs can either resolve to IP addresses (inducing what we call *EID-to-IP mappings*) or to one or more EIDs (inducing *EID-to-EID⁺ mappings*). If the Target field of the **erecord** contains only an IP address i , then, as described in §3.2, the sender simply transmits a packet whose destination IP address is i and whose destination EID is e . In this case, the EID owner may or may not be directing potential senders to a delegate, but the semantics are the same: the EID owner is saying “to reach me, send your packet there”.

If, on the other hand, the Target field of the **erecord** contains one or more EIDs, then the recipient is expressing its wish that the packet transit one or more intermediaries before reaching the recipient. In this case, the semantics are “to reach me, send your packet to these intermediaries in sequence”. The sender would resolve the first EID in the series to an IP address j (perhaps via intermediate resolutions to other series of EIDs, each of which would be injected into the original series in the logical order) and send the packet to j . This stack of EIDs is carried in the DOA header; transport connections are bound to the last EID, which identifies the ultimate destination. (The design, but not our implementation, lets an EID resolve to multiple IP addresses; the multiplicity reflects a multi-homed host or an anycast situation in which a set of hosts are equivalent for the **erecord** owner’s purposes. Similarly, each EID in the Target field could really be a set of EIDs, again representing equivalent hosts.)

To send a packet back to the source, the receiver executes the steps just described to resolve the sender’s EID, f . The receiver cannot simply use the source IP address in the original packet as the destination IP address in the reply packet because f may resolve to a different IP address (e.g., f ’s host sends packets directly but wants packets to it sent through an intermediary).

To spare the server the burden of a DHT lookup, the client can send its **erecord** as an optimization. (The client may have to send more than one **erecord** since the client’s EID may resolve to a chain of EIDs before being resolved to the IP address needed by the server.) Also, DOA hosts use the **erecord**’s TTL to maintain a TTL-based cache of EID-to-IP and EID-to-EID⁺ values, thus avoiding a DHT lookup for every packet.

The **erecord** and accompanying machinery exist to support receiver-invoked intermediaries. *Senders* invoke additional intermediaries by pushing the EIDs of the intermediaries onto an identifier stack in the DOA header.

4.3 Security and Integrity

Because identities (namely, EIDs) are separate from locations (namely, IP addresses), the following requirement arises under DOA: *The mapping from a given EID to its target must be correct, i.e., either resolving an EID, or using an erecord directly sent by a host, must yield the IP address intended by the EID owner or by the EID owner’s delegates.* Specifically, DOA must satisfy the following properties:

1. Anyone fetching an **erecord** must be able to verify that the EID owner created it.
2. Only the owner of an EID may update the corresponding **erecord** in the DHT.
3. When a host sends its **erecord** to another host without using the DHT, the sending host must not be able to forge the **erecord**.

To uphold these properties, DOA uses self-certification [36]: EIDs must be the hash of a public key, and the **erecord** is signed with the corresponding private key. When a host either performs a `get()` operation on the DHT, resulting in an **erecord**, or else receives an **erecord** directly from a purported EID owner, the host must check that the **erecord** is signed with the private key whose corresponding public key was hashed to create the EID in question. DHT nodes also perform this check before accepting **erecords**. For more details, including how EID owners may update their public keys without changing their EIDs, see [61]; we adopt the mechanisms described there.

With the above properties satisfied, **erecords** cannot be forged, but senders can still spoof source EIDs (i.e., put the wrong source EID field in the packet). This attack is like spoofing a source IP address today (except that ingress and egress filtering, which help guard against IP address spoofing, are not applicable to EIDs): successful attacks do the same damage, and both attacks are detectable under two-way communication. For example, if a TCP client tries to spoof a source EID to a TCP server, when the server looks up the source EID (or uses the signed **erecord** supplied by the client), the server gets the *correct* (not fake) IP address for that EID, so when the server replies to the IP address, the host at that address will not complete the 3-way handshake.

Security of the DHT itself is a topic outside the scope of this paper. We briefly observe that DHT nodes cannot forge **erecords** but can return old versions of **erecords**. A way to guard against this attack by consulting multiple DHT nodes, instead of one, is mentioned in [14].

Also, we note that while IP source routing creates security problems, DOA’s loose source routes of EIDs do not inherit these difficulties. With IP source routing, receivers reverse the source route to reply to a sender, which allows an adversary to carry out a man-in-the-

middle attack by placing its IP address in a forged source route. Under DOA, however, hosts do not reverse the loose source route to reply to a sender.

4.4 Host Software

We now describe the software interface that a production DOA deployment would expose. Our prototype implementation differs from this description; see §7.1.

DOA software would run in the kernel and be exposed to applications with the Berkeley sockets API [37], which can extend to EID-based identification. Applications would open a new socket type, PF_DOA (in analogy with PF_INET, used by today's IPv4-based applications), and pass to the API a new data structure, the `sockaddr_eid`, which holds an EID and port (just as the `sockaddr_in`—which today's IP-based applications use—holds an IP address and port). Some of the socket calls, such as `connect()` and `sendto()`, might cause the DOA software, depending on the state of its caches, to issue one or more DHT lookups to resolve the EID into potentially intermediate EIDs and also an IP address. One could port today's applications by substituting `sockaddr_eid` for `sockaddr_in` in the code, though client applications would need additional logic to get a server's EID, perhaps via a DNS lookup.

For example, client TCP applications would call `connect()`, supplying a `sockaddr_eid` that contained an EID and port, both of which the application had obtained out of band. Similarly, TCP server applications would call `accept()`, getting back the EID and port of the initiating client. To reply to the client, the server's DOA software would resolve the client's EID to an IP address *i* and address reply packets to *i* at the IP layer.

For bootstrapping, DOA hosts would be configured with the EIDs and IP addresses of one or more of the DHT nodes, in analogy with how today's hosts learn the IP address of a DNS resolver (via hardcoding or DHCP). On boot up, the DOA software would insert into the DHT the host's `erecord` (which could contain an EID-to-EID⁺ or EID-to-IP mapping, depending on the host's configuration) and would refresh the mapping periodically or in response to host configuration changes.

4.5 Limitations

DOA hosts cache `erecords`, so hosts may have stale information about prospective peers. Also, two DOA peers in a TCP session resolve each other's EIDs only once—at the start of the session—so hosts cannot change locations without breaking existing connections. DOA could overcome this limitation if it were extended with a signaling mechanism, as in [39, 53], that allows hosts to notify peers of IP address changes. Finally, an EID owner cannot change which intermediaries are invoked based on who is trying to communicate with it.

5 Network Extension Boxes Under DOA

This section and the next describe example intermediaries under DOA. In the next section (§6), our focus is on filtering packets and how to move this function “off-path”. In this section, we show how the DOA framework accommodates boxes that bridge between different IP address spaces and also simplifies the use of these boxes. Under the status quo, these boxes are known as NATs but would be reincarnated under DOA as tenet-upholding Network Extension Boxes (NEBs).

We first consider three usage scenarios for NEBs (§5.1), then give our general approach, including a short discussion of architectural coherence (§5.2), and then discuss the benefits of this approach (§5.3). One of the benefits, automatically exposing hosts behind NEBs, is particularly useful when NEBs are cascaded (§5.4). We present several mechanisms for achieving automatic configuration (§5.5) and require that they work when there are multiple levels of NEB. We conclude the section with a discussion (§5.6).

5.1 Scope

The following NEB scenarios reflect reasons for deploying NATs today (§2.1) and are ordered by the degree of access control:

- (a) A host behind the NEB is accessible on all ports. The NEB creates a separate addressing realm but does not control access. Under this scenario, which corresponds to the “Convenience and Flexibility” reason for deploying a NAT today (§2.1), many hosts within an organization can be reachable as first-class members of the Internet, even if the organization has only one IP address.
- (b) A host behind the NEB is accessible on configured ports, and the NEB blocks unsolicited traffic to the host on the other ports. This scenario, which reflects both reasons for deploying a NAT (§2.1), is analogous to, *e.g.*, today setting up a Web server behind a NAT and configuring the NAT to send all packets with destination port 80 to the Web server.
- (c) A host behind the NEB is accessible on no ports, *i.e.*, the host can only receive packets associated with connections it has initiated. This scenario, which is principally driven by the “Security” reason for deploying a NAT (§2.1), is the default under NAT today.

We expect that under DOA, scenario (b)—a mix of access control and exposure—would be most common. However, for clarity, we focus on scenario (a) and return to scenarios (b) and (c) at the end of the section (§5.6).

5.2 Approach

NEBs preserve packets' DOA headers and use the destination EID field as a demultiplexing token. For example,

the NEB could maintain an EID-to-IP table, look up the destination EIDs of incoming packets, and then use the results of these lookups to rewrite the destination IP addresses. There are other ways to demultiplex; we cover them in §5.5.

This approach upholds the two tenets stated earlier. Tenet #1 holds because an end-host behind a NEB can pass its EID to others, who can then use this handle to direct packets to the given host. As mentioned in §3.3, to obey network-level layering (tenet #2) NEBs may only rewrite fields in a packet if the packet is addressed to the NEB. Since NEBs, like today's NATs, have to rewrite both the destination IP addresses of inbound packets (to demultiplex them) and the source IP addresses of outbound packets (to make them appear as if they originated at the NEB), the discussion in §3.3 implies that both inbound *and* outbound packets be addressed to the NEB at the IP layer.

However, this approach, in pure form, makes the NEB resolve the destination EIDs of outbound packets. As a practical matter, sources of outbound packets could do the resolution and put the resulting IP address somewhere in the packet, thereby sparing the NEB this resolution burden. The source could even put the resulting IP address in the destination IP address field; at the IP layer, then, outbound packets would look alike under NEB and NAT. This modified approach—which technically violates the rules in §3.3 but is consistent with the spirit of the tenets because the violation is under the control of the end-host—is what we adopt.

5.3 Benefits

Upholding the two tenets results in the following benefits, some of which solve the problems stated in §2.1.

End-to-end communication. Communication is logically between two EIDs. Thus, protocols can uniquely identify hosts.

Ports are not overloaded. Not using the destination port as a demultiplexing token lets multiple hosts behind a NEB receive packets sent to the same destination port.

VPNs. Getting VPNs to work through NATs is cumbersome and complicated [44]. The difficulties under the status quo result from NATs rewriting both ports and IP addresses. Under DOA, NEBs do not rewrite ports, and the state associated with encrypted tunnels could be bound to EIDs, not IP addresses.⁶

Automatic configuration. Under DOA, the process of exposing a host behind a NEB can be automated. When NEBs are cascaded, a scenario covered in the next section, this automation is particularly useful—and particularly problematic under the status quo (§2.1).

⁶Much of the HIP work [40] focuses on such binding of IPSEC state to cryptographically imbued EIDs.

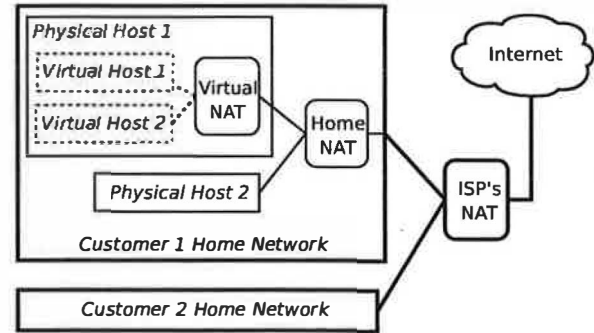


Figure 4: A tree of NATs.

5.4 Cascaded NEBs

The scenario of multiple address realms between a given host and the rest of the Internet is becoming more frequent. Consider the following example, depicted in Figure 4: an individual runs a virtual host (using, e.g., VMWare [60]) that runs behind a NAT on the physical host (such NATing of virtual hosts is common). The physical host is in turn a member of a home network that is all behind a single NAT, which is connected to a broadband provider. The link from the broadband provider, owing to the provider's operations, is itself NATed, making, altogether, three levels of NAT between the virtual host and the global Internet.

We now cover protocols for automatically configuring NEBs to expose servers; we require the protocols to work when servers are behind multiple levels of NEB.

5.5 Secure Automatic Configuration

A protocol for configuring NEBs to expose servers must satisfy three requirements. First, the protocol must tell the end-host what to put in its record since an end-host separated from the global Internet by levels of NEB has no *a priori* knowledge about the IP addresses of NEBs between that end-host and the Internet. Second, the protocol must establish state, either in NEBs or in the EID resolution infrastructure, that allows NEBs to use the destination EID field in packets as a demultiplexing token for rewriting the destination IP address field.

Third, this state must correspond to the wishes of the actual EID owner, rather than of an impostor trying to divert the EID owner's traffic. This focus on authenticity is warranted because passing unprotected protocol messages through levels of NEB could be problematic. For example, an upstream provider cannot trust NEBs administered by its customers, and end-users cannot trust each other's NEBs to correctly propagate control or data messages. Also, NEB networks, like today's NATs, would often be constructed over wireless links, which are susceptible to eavesdropping and tampering. In what follows, we assume that a NEB trusts only the NEB directly upstream of it (called its *parent*); that NEBs

and end-hosts know the EID of their parent; and that all links in the NEB network are vulnerable to eavesdropping, tampering, and arbitrary data injection.

We now give three mechanisms, each using a different kind of EID resolution, that meet the requirements above. We implemented the third one; see §7.2.

5.5.1 EID maps to EID

Each NEB and end-host creates a mapping in the global EID resolution infrastructure from its EID to its parent's EID; in other words, NEBs and end-hosts use the delegation primitive to say, "to reach me, send your packet to my parent's EID". Also, each NEB holds a mapping from its children's EIDs to its children's internal IP addresses.

Control plane. Assume an end-host with EID e_0 must traverse NEBs with EIDs e_1 through e_n before reaching the Internet. The end-host inserts a mapping from its EID (e_0) to its parent's EID (e_1) into the global EID resolution service. The end-host also sends a message to e_1 informing it of a mapping between its EID (e_0) and its IP address (i_0). All other internal NEBs in the chain (e_1 through e_{n-1}) use the same protocol. The outermost NEB uses the global EID resolution infrastructure to map its EID (e_n) to its IP address (i_n), which is globally reachable. A NEB with EID e_{j+1} should only accept an EID-to-IP mapping of the form $\langle e_j, i_j \rangle$ if the mapping is authentic, *i.e.*, if it is signed by the private key corresponding to e_j ; performing this check might require e_j to send e_{j+1} its public key (which should hash to e_j).

This approach, as just described, is vulnerable to replays of $\langle e_j, i_j \rangle$ mappings. Such replays would allow the wrong end-host—one that is later assigned IP address i_j —to redirect e_j 's traffic to it. We show how one might protect against these attacks in §5.5.3.

Data plane. Assuming the end-host and intermediate NEBs all initialize successfully, a remote client can send data packets to the end-host (with EID e_0) by using the EID resolution infrastructure to map e_0 to e_1 , e_1 to e_2 , and so on, up the NEB chain. The last EID lookup maps e_n to the IP address i_n . The client then stacks the identifiers e_0 through e_n in its packets and sends the packet to IP address i_n . Once the packet reaches the outermost NEB (e_n), the NEB pops the top EID off the stack to find that e_{n-1} is the packet's next hop. The NEB then consults its routing table to map EID e_{n-1} to IP address i_{n-1} , rewrites the packet's destination IP address to i_{n-1} , and forwards the packet. This process continues until the packet reaches its eventual destination, e_0 .

5.5.2 EID maps to EID and a Hint

Another approach uses the *erecord*'s Hint field, mentioned in §4.2, to relieve NEBs of state.

Control Plane. The end-host inserts into the EID resolution infrastructure a mapping from its EID, e_0 , to the EID, e_1 , of its parent NEB; the *erecord* holding this

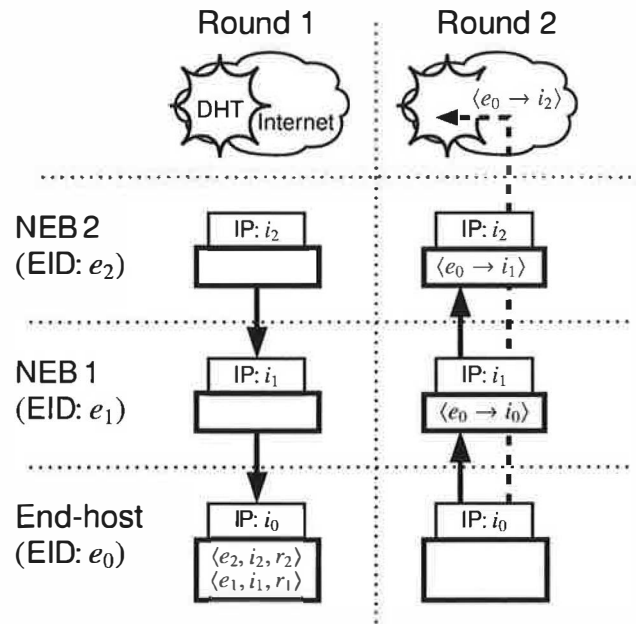


Figure 5: NEB and DHT state after each DOA-RIP round.

mapping has in its Hint field the end-host's internal IP address, i_0 . The NEB e_1 likewise creates a mapping in the EID resolution infrastructure from its own EID to the EID, e_2 , of its parent and puts its "outer" IP address, i_1 , into the Hint field of the *erecord*. This process continues until the outermost NEB inserts a mapping from its EID, e_n , to its "outer" IP address, i_n .

Data plane. A remote host wishing to communicate with e_0 resolves e_0 to e_1 , e_1 to e_2 , ..., e_{n-1} to e_n , while remembering the Hints i_0, i_1, \dots, i_n . As with the previous mechanism, the remote host stacks the identifiers e_0 through e_n in its packets—and in this case also includes in the DOA header the IP addresses i_0 through i_n —then sends the packet to IP address i_n . Once the packet reaches the outermost NEB (e_n), the NEB pops the top EID and IP address off the stack to find that e_{n-1} with IP address i_{n-1} is the packet's next hop, and the process continues.

5.5.3 EID maps to IP address

The previous two mechanisms require a prospective sender to do as many EID resolution infrastructure lookups as there are levels of NEB. An alternative, that we call DOA-RIP, allows senders to do a single resolution: from the EID, e_0 , of the end-host to the IP address, i_n , of the outermost NEB.

Control plane. End-hosts and NEBs follow a two-round protocol, depicted in Figure 5. In the first round, the end-host (with EID e_0) sends an initialization message to its parent in the NEB tree; intermediate NEBs forward the message until it reaches the outermost NEB (with EID e_n). The outermost NEB creates a message

$x_n = \langle e_n, i_n, r_n \rangle$ (r_n is a random nonce to prevent replay attacks), signs x_n , and sends it to the NEB with EID e_{n-1} . Each NEB e_k ($k < n$), follows suit, appending the message $x_k = \langle e_k, i_k, r_k \rangle$ to x_{k+1} . When the end-host receives x_1 , it verifies the message using e_1 's public key. This message is a *route* to the global Internet.

In the second round, the end-host creates a series of requests $y_k = \langle e_0, i_k, r_k \rangle$ for $1 \leq k \leq n$, signs each y_k individually; concatenates all the y_k 's and appends its public key; and sends this message up the NEB chain. Each NEB e_k verifies y_k using e_0 's public key and signature. Each NEB further checks that r_k is in its cache and that r_k is the nonce it issued in the first round for EID e_0 (the NEB flushes r_k from a cache within a fixed number of seconds—10, in our implementation—of issuing r_k). If these checks succeed, the NEB flushes r_k , establishes a mapping $\langle e_0, i_k \rangle$, and propagates the request up the NEB tree. If all NEBs successfully establish the mapping, the end-host inserts into the EID resolution infrastructure a map from e_0 to i_n .

Data plane. To communicate with the end-host, remote clients first resolve e_0 to i_n and then send packets with destination IP address i_n and destination EID e_0 , at which point the outermost NEB, and all succeeding NEBs in the chain, use their internal state to forward the packet to the end-host.

5.6 Discussion

Other scenarios. Though we focused on scenario (a) (from §5.1), the benefits noted above (in §5.3) apply equally to scenario (b). Two of the three mechanisms for automatic configuration also apply (the stateless NEB from §5.5.2 does not) with the one change that end-hosts—when making signed requests of parent or ancestor NEBs to add EID-to-IP mappings—need to add requests to open (or block) specific ports. This type of automatic hole punching works under DOA, in contrast to the status quo, for three reasons: (1) DOA has a persistent notion of host identity, which allows NEBs to associate policies with hosts and remote network entities to identify hosts behind the NEB; (2) port fields are not overloaded under DOA, so internal nodes in the NEB tree do not have to coordinate among themselves, in contrast to the status quo wherein only one server in a tree of NATs can receive, e.g., traffic destined to port 80 on the outermost NAT's public IP address; and (3) hosts can leverage the cryptographic properties of their identities to create signed messages saying “handle my packets like this”.

The benefits above, except automatic configuration, also apply to scenario (c). Although this scenario is the strictest access control NEBs offer, network administrators may still prefer NATs, since NATs, unlike NEBs, obscure the identities of the organizations' end-hosts. Our response is that organizations today use NATs in

part because they hide internal network topology. Since EIDs are independent of network internals, organizations might be looser about exposing EIDs than IP addresses.

Comparison of the mechanisms. Observe that the three mechanisms above are different ways to perform routing that offer different trade-offs between state held in the NEB and the degree of fate-sharing. With one of the mechanisms (§5.5.2), all information about EID-to-IP mappings is in the EID resolution infrastructure, which simultaneously frees the NEB of state but makes correct routing depend on the availability of the resolution infrastructure. In contrast, DOA-RIP pushes nearly all state into the NEBs along the path between two communicating entities.

6 Network Filtering Boxes Under DOA

In this section, we demonstrate DOA's delegation primitive with a simple remote packet filter (RPF) box that yields functionality similar to today's firewalls but need not be interposed between a host receiving firewall service and that host's link to the Internet. One can certainly get similar functionality today with special-purpose machinery (e.g., VPN software, though their interfaces differ across solution providers). However, we believe that decoupling services from topology is best done with *architectural*, rather than *application*, support because: (1) users should be able to compose intermediaries and (2) users should be able to change their delegates easily (see §3.4), both of which imply that the architecture support a standard, application-independent invocation method.

6.1 Approach and Design

The RPF is a basic application of DOA's mechanisms; it is depicted in Figure 6. A user (or representative of the user, e.g., corporate IT staff) wanting remote firewall service creates a mapping in the EID resolution infrastructure from the end-host's EID, e , to the RPF's EID, f (or to the RPF's IP address, but then if that IP address changes, the resolution of e will be incorrect). This end-host expresses its actual network location either by putting its IP address, i , in the Hint field of the `erecord` to which e resolves, or by communicating directly with the RPF and telling it about the association between e and i . (Our implementation, described in §7.3, uses the second option.)

When a sender attempts to contact e , it first looks up e in the EID resolution infrastructure, sees that e maps to f , and then further resolves f to an IP address (which might involve intermediate resolution steps, depending on whether the RPF itself has delegates). In the simple case in which f resolves directly to an IP address j , the sender forms IP packets with destination address j and destination EID e . Note that f must be in the *stack* of identifiers since the host given by j may actually be the RPF's *delegate* rather than the RPF itself (e.g., if the RPF

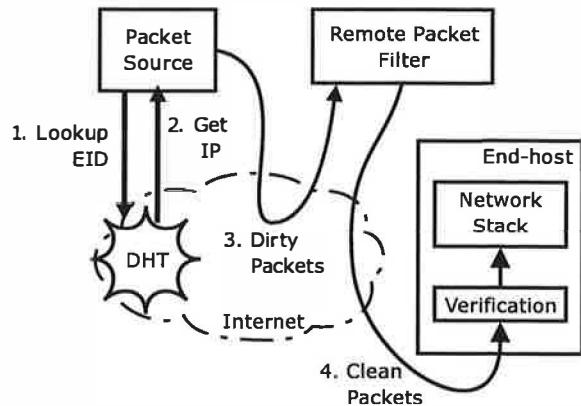


Figure 6: Packet filtering under DOA using delegation. End-hosts apply a simple verification rule, not a collection of them.

were behind a NEB, the NEB would need f 's EID to make a decision about the packet).

When receiving IP packets, the RPF extracts the destination EID e from the DOA header, looks up the set of rules associated with e , and finally applies these rules; examples of such rules are filters to block or accept traffic based on IP- or transport-layer fields. The result is “passing” or “failing” a packet. When packets “fail”, the RPF drops the packet.

The RPF attests that packets “passed” by inserting into the packet a MAC (Message Authentication Code) taken over the packet; the MAC is keyed with a secret shared between RPF and end-host. The RPF then rewrites the packet's destination IP address and sends the packet to the end-host, which applies a single rule: redoing the MAC computation and testing whether the result matches the MAC in the packet. The end-host ignores packets that fail this test; thus, only packets that have been vetted by the RPF are processed by the end-host's networking and application software. The MAC is carried in a DOA security header, which extends the standard DOA header described in §4.1.

The RPF depends on both of DOA's core mechanisms: first, because of unique host identifiers, the RPF has a way (namely the destination EID field) to distinguish among hosts, allowing it to apply host-specific rules and then send the processed packet to the correct destination. Second, the delegation primitive is what allows the RPF to be invoked in the first place. See §7.3 and §8.3 for implementation and evaluation details.

6.2 Benefits

We first claim two architectural benefits, as discussed in §3.3 and §3.4: the RPF described here does not violate network-level layering, and also, a market for such services could arise.

These architectural benefits lead to simplification for users. Getting firewall rules right is hard, far beyond or-

dinary users' ability, and commercial products (*e.g.*, Norton [59]) require users to keep their software current. Outsourcing per-packet rules to a central provider solves those problems. Of course, end-hosts still have to check packets, but the check—“was this packet vetted by my RPF provider?”—is considerably simpler than the usual complement of firewall rules.

6.3 Limitations

The box just described is primitive. For it to provide the same functions as today's firewalls—such as using existing TCP connections, and not just stateless filtering rules, to make decisions—protected end-hosts would have to direct their outbound traffic through the RPF. These end-hosts would use the mechanism of sender-invoked intermediation (§4.2).

6.4 Physical Security

Some organizations require that every inbound and outbound packet be vetted by a box that is physically interposed between the organization and its link to the Internet. We briefly describe two scenarios for such on-path boxes under DOA.

We start with an on-path vetter that works with an off-path RPF. As above, an end-host within the organization, h , creates a mapping in the EID resolution infrastructure from its EID, e , to the RPF's EID. In this case, however, h tells the RPF that after the RPF processes packets destined to e , it should send them to the vetter's IP address (instead of to h 's IP address, as above). The vetter allows packets into the organization only if they are addressed to it at the IP layer and if the MAC check succeeds, thereby ensuring that the RPF has checked every packet entering the organization. The vetter uses the destination EID field to forward vetted packets to the correct host.

Some organizations will of course not want an RPF, preferring to deploy an on-path firewall and manage the rules itself. DOA supports an on-path firewall just as it does an off-path firewall: the organization's hosts map their EIDs to the EID or IP address of the on-path firewall. Since this setup is functionally the same as today's on-path firewalls that are not explicitly invoked at the IP layer, one might wonder what DOA accomplishes here. The answer is uniformity: in this setup, the configuration of end-hosts is independent of the firewall's placement. Thus, administrators can later move the firewall off-path without reconfiguring every host in the organization.

7 Implementation

We describe our implementation of end-host DOA software, a NEB prototype, and an RPF prototype.

7.1 End-Host DOA Software

In a production deployment, DOA software would be part of kernel protocol stacks, as in §4.4. However, we

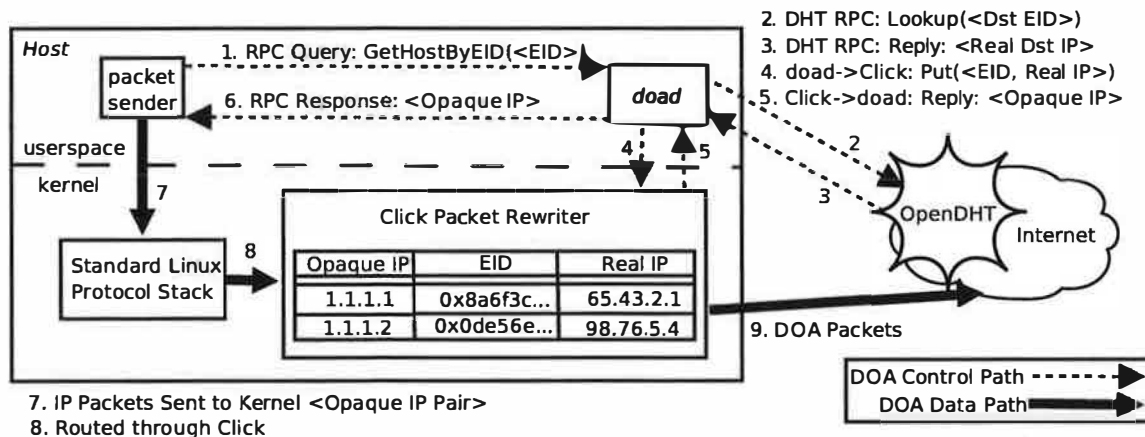


Figure 7: The control and data paths in our prototype implementation of DOA. This figure depicts sending a packet to a given EID obtained out-of-band. Events are numbered in chronological order.

wanted to understand DOA’s properties before committing to full kernel implementations, and so we prototyped using a combination of user-level software and Click [31] modules inside the Linux 2.4.20 kernel. Beyond a patch required by Click, we did not modify the kernel. Figure 7 depicts our implementation of end-host DOA software.

Applications get EIDs via user input or DNS and resolve them by invoking the `GetHostByEID` RPC, exposed by `doad`, which is a user-level daemon written in C++ using the SFS `libasync` library [35]. `doad` implements the RPC by first querying OpenDHT [29] (the key is the EID, e , the returned value is the IP address, i , that e resolves to) and then, via Click’s `/proc` file system interface, telling the Click “rewriter” module about the mapping $\langle e, i \rangle$. `doad` returns an opaque handle in the 1.0.0.0/8 subnet that the application uses when the sockets API expects an IP address; the opaque handles allow us to reuse much of the kernel’s IPv4, TCP, and UDP software. (Applications could use EIDs instead of the opaque handles if the kernel’s networking software were extended to use the `sockaddr_eid` structure, as described in §4.4.) The rewriter module receives IP packets with addresses in the 1.0.0.0/8 subnet, maps these opaque handles to EIDs and real IP addresses and then transmits bona fide DOA traffic. We did not implement EID-to-EID⁺ mappings.

7.2 NEB Prototype

We implemented (1) a NEB prototype in a Click module and (2) DOA-RIP (§5.5.3) in user space. The NEB has an EID-to-IP table which it uses to rewrite destination IP addresses and which gets an entry when a host behind the NEB, possibly separated by several other NEBs, runs DOA-RIP. After DOA-RIP completes and the NEBs, which also run DOA-RIP, have correct state, the host uses `doad`’s interface to OpenDHT to insert a mapping from the host’s EID to the outermost NEB’s external IP address, thereby making the host globally reachable.

7.3 RPF Prototype

The RPF is (1) a Click module that associates EIDs to a set of simple rules that are together applied (with OR or AND) to IP-, DOA-, and transport-layer fields to make a “pass” or “fail” decision for each packet and (2) user-level software that communicates with end-hosts, first, to establish a secret key for each EID (using an encrypted, MACed control channel given by the SFS [36] toolkit) and, second, to process requests to add, change, or remove rules. End-host RPF users run (1) a MAC-checking Click module that injects into the kernel’s networking stack only those packets that have been correctly MACed by the RPF and (2) user-level software to communicate with the RPF, as just described. The RPF uses HMAC [32] and, in our prototype, it is taken over packet headers, only.

8 Evaluation

The architectural coherence afforded by DOA comes at a performance cost. This section characterizes that cost with microbenchmarks that measure the latency, throughput, and processing time overhead of DOA-enabled data transfers.

8.1 Round-trip Times and Hops

Compared to the status quo, DOA adds network round-trip times. DOA requires an extra resolution—of the EID—when a host first sends a packet to another host. For applications whose end-to-end latency is dominated by DNS lookups (such as Web browsing), the effect of these resolutions might be particularly pronounced. In the most basic DOA configuration—two end-hosts communicating with no off-path intermediaries—the connecting client makes two synchronous network calls: (1) a DNS request to map a human-readable hostname to an EID and (2) a DHT lookup to map a server’s EID to its IP address. A third synchronous DHT lookup is required for the server to resolve the client’s EID to an IP address.

A recent study [26] indicates that median DNS lookups from a network at MIT can vary from about 70 ms in the case of NS server cache hits, to about 190 ms in the case of cache misses. By contrast, we measured median DHT lookups of random EIDs stored in OpenDHT at 138 ms. Thus, DOA can add noticeable delays to small data transfers, sometimes tripling their end-to-end latencies.

However, for latency-sensitive hosts and applications, the following optimizations are possible:

- The DHT could use Beehive’s [45] proactive, model-driven caching strategy to reduce the number of network round-trips required by lookups to an average of one or less than one (assuming the request pattern for EIDs is heavy-tailed).
- DNS names of hosts could resolve to the EID *and* the erecord (or to the chain of erecords that together indicate how to reach the host), thereby requiring one DNS lookup, as under the status quo, to send a packet to a host. In this case, DNS itself would be caching erecords.
- To save a remote host the burden of an EID resolution when responding to an initiating host, the initiating host could send its erecord (as noted in §4.2).

In addition, DOA adds network hops: when a packet travels from a source to an off-path middlebox en route to a destination, the packet (in most cases) takes more network hops than if it had traveled from source to destination directly. This extra latency is inevitable if one wants to invoke an off-path intermediary. There is no “correct” trade-off between latency and the flexibility of off-path functions; different users have different preferences.

8.2 Packet Size Overhead

DOA packets in our implementation have a 68-byte DOA header (the 44 bytes shown in Figure 2 plus 24 for the DOA security header mentioned in §6.1). This overhead affects the maximum number of packets per second sustainable by DOA senders and receivers and is more costly for smaller packet payloads. For example, adding a DOA header to a 1466-byte UDP-over-IP-over-Ethernet packet (the UDP payload here is 1400 bytes) increases the packet size by 4.6%. For 130-byte packets (with UDP payload of 64 bytes), the 68 bytes of DOA header add overhead of more than 50%.

For large packets, this overhead is likely to bottleneck DOA’s sustainable throughput. To verify this claim, we now characterize the throughput our DOA implementation can sustain when sending and receiving large packets. We measure both DOA and non-DOA traffic and find that the packet header overhead introduced by DOA explains the throughput difference between the two cases. Each measurement below is the average of five trials involving 1 GB of data, and the average packet drop rate

Component	cycles/pkt	μ s/pkt
DOA→IP	1894.2	1.11
filter	9410.3	5.54
verify	8773.8	5.16

Table 1: Processing time per packet for DOA components. The first column contains the number of cycles, while the second column contains the calculated time (in μ s) needed to perform that number of cycles on an Intel Celeron 1.7 GHz processor.

was less than 1%. Our experiments do not involve the DHT; prior to the experiments, we resolved the destination EIDs to IP addresses.

To get a baseline, we measured the number of UDP packets per second that one of our test hosts can send another. We tested large packets (1400-byte payloads) over a Gigabit Ethernet network, tuned the sending rate to achieve maximum throughput, and measured the number of packets that exited the receiver’s device driver queue. On average, the receiver processed 72900 packets per second, or 778.5 Mbit/s. The bottleneck here appears to be our hosts’ PCI buses.

Next, we ran the same test with DOA packets. The sender uses our end-host DOA software (§7.1), which inserts a DOA header into IP packets and rewrites IP headers. The receiver performs a similar process to translate DOA packets to IP packets. In this case, the receiver processed 69600 packets per second, or 743.0 Mbit/s, which is 4.6% slower than the baseline. We conclude that the slowdown here is due entirely to packet size overhead. These tests were for large packets; as noted above, small packets will be much more penalized by this overhead.

8.3 Processing Time

For small packets, however, in addition to packet size overhead, CPU costs for per-packet DOA operations will limit the rate at which DOA hosts can process packets. We now characterize this potential bottleneck on small packets (64-byte UDP payloads). Using Click tools to read the processor’s cycle counter before and after our DOA modules, we estimated the number of cycles used by the modules; the reported numbers are averages over 80000 processed packets. Note that these numbers are upper bounds on average processing time: the implementation is untuned, and our measurements include cycles consumed by interrupt handling for other kernel processes. Table 1 summarizes our observations.

We first measured the processing time needed by our receiver’s DOA software, which translates DOA packets to IP packets (labeled “DOA→IP” in Table 1). This component takes nearly 1900 cycles—or 1.11 μ s on the host we used for testing, which has an Intel Celeron 1.7 GHz CPU—to process each packet.

We next considered the processing time for the operations associated with an RPF (§6), namely HMAC [32]

computation and verification. In our experiments, the RPF operates as described in §7.3; it uses a single default rule that passes all packets intended for the receiver and holds a mapping from the receiver's EID to an IP address. The RPF computes the MAC for each packet, writes the MAC to the DOA header, and forwards the packet to the receiver. When the receiver gets the packet, it verifies the MAC before passing the packet to its end-host DOA software. As shown in Table 1, the filter takes more than 9400 cycles ($5.54\ \mu\text{s}$) to apply its rule and compute a MAC, and the receiver takes nearly 8800 cycles ($5.16\ \mu\text{s}$) to verify the MAC.

8.4 Discussion

Of the three types of costs imposed by DOA—lookup latency, packet size, CPU overhead—the latter two would only appear to users under the most stressful system conditions. The first cost, latency, is serious because, absent optimizations, it is visible to end-users. However, as noted in §8.1, there are several caching strategies that substantially mitigate, if not eliminate, this latency. Ultimately, we believe that the costs imposed by DOA are outweighed by the benefits of a coherent framework for reasoning about, and deploying, intermediaries.

9 Related Work

Besides the direct influence of i3 [57], HIP [39–41], and UIP [17] on our mechanisms and insights, an older proposal for location-independent EIDs [34] grew out of Nimrod [9]. Shoch [52] and Saltzer [50] have been among many (see [10, 11, 13, 19, 33, 42, 43] and references therein) to distinguish between network elements' identity and location. Indeed, much of what we mention below separates these two concepts, usually by creating a set of end-host identifiers distinct from network location.

i3 canonizes this separation with an infrastructure that uses flat identifiers in packets to decouple sending (into the infrastructure) and receiving (from the infrastructure). These identifiers name services whereas EIDs name hosts. Like DOA, i3 is specifically designed for senders and receivers to invoke intermediaries. i3 does not hold the proliferation of private addressing realms as a principal concern, but one can leverage i3 to reach machines behind NATs without modifying or configuring the NATs [28]. The main difference is that the DOA architecture requires a *resolution* infrastructure while i3 depends on a *forwarding* infrastructure; under the pure design, all i3 packets are sent into the infrastructure.

TRIAD [22] is an extension to the Internet that addresses many architectural ills, including NAT. TRIAD hosts receive location-independent names. As in DOA, these names may resolve to a logical path, and IPv4 addresses are routing tags without end-to-end significance. TRIAD does not focus on a framework for network-layer middleboxes, though its mechanisms can certainly ac-

commodate them, and the authors give a solution for NAT traversal. The technical details of our approaches differ: TRIAD names are derived from domain names (in contrast to flat EIDs), and under TRIAD, resolution and routing are conflated, thereby improving latency.

HIP also separates location and identity; its goal is architectural support for mobility and multi-homing. DOA borrows some of HIP's mechanisms and applies them to middlebox issues, which is not HIP's focus.

In contrast, some work is expressly motivated by the proliferation of private addressing realms. UIP [17], from which we also borrow, creates an overlay among participating hosts to interconnect heterogeneous or NATed networks. Like DOA, UIP incorporates HIP-style flat host identifiers. UIP hosts form an ad-hoc overlay by using a DHT-inspired algorithm to route packets for each other based on the destination identifier. Our approach contrasts with UIP's in that, while both projects address middleboxes' proliferation, we focus on an architecture that explicitly welcomes middleboxes whereas UIP's overlay of peers makes them transparent. IPNL [20] is an extension to the Internet architecture that solves problems resulting from private addressing realms. IPNL relies in part on bona fide host identifiers; these identifiers are domain names, though the authors acknowledge the security benefits of HIP-style flat identifiers. Like DOA, IPNL tries to coherently incorporate NATs into the Internet architecture, and both designs modify hosts and NATs but not IPv4 routers.

Other projects have tried to obsolete middleboxes; these run the gamut from architectural enhancements to radical reorganizations. An example in the middle of this spectrum is IPv6 [15]. IPv6 addresses are globally unique (thus addressing one motivation for NATs), but, as noted in §3.2, do not satisfy the requirement of topology-independence. Predicate Routing [47] and network capabilities [1] propose architectural enhancements for security and denial-of-service protection. Radical network architectures and meta-architectures include Role-Based Network Architecture [7] and FARA [10]. Our approach contrasts with these because, first, our goal is explicit invitation of middleboxes and, second, these proposals, if fully realized, require at least some changes to all network elements, not just hosts and middleboxes.

In contrast, other work has avoided creating identifiers for end-hosts but has nonetheless accepted middleboxes as an architectural problem to be worked around. MIDCOM [56, 58] is a protocol and framework intended to remove intelligence from NATs and firewalls by offloading application-specific behavior to designated agents, which insert dynamic state into intermediaries automatically. For example, in response to a globally reachable host initiating an Internet telephony session to a NATed host, the agent would ask the NAT to open the appro-

priate destination UDP port and would close the port at session's end. Like DOA, MIDCOM aims to simplify management of NATs and firewalls by creating state automatically. However, because MIDCOM focuses only on application- and not networking- and transport-level software, persistent host identifiers are unavailable to them, and thus their protocols devote considerable energy (and complexity) to handling the overloading of port fields. Also, MIDCOM's techniques work through only one layer of NAT [56] in contrast to our supposition that hosts may be behind several layers.

Twice NAT [55], Realm Specific IP [6, 55], and STUN [48] all address specific problems posed by NATs. A recent Internet draft [18] summarizes various techniques by which P2P applications can handle middleboxes. While useful for the current network architecture, these (largely manual) tactics for exposing NATed hosts would be unnecessary if all hosts had location-independent identifiers. Today, many home users attempt to create persistent identifiers for frequently renumbered hosts with Dynamic DNS, *e.g.*, [16]. Since DNS names are resolved to IP addresses and are not carried in packets, they are quite useful as naming handles for humans but not for network elements.

DOA's use of the delegation primitive to simplify firewalls is preceded by a body of literature that addresses the error-prone and time-consuming nature of firewall configuration. The Firmato toolkit [4], for example, takes a language-based approach to simplifying firewall configuration by abstracting away low-level configuration details. Distributed firewalls [5,25,30] take simplification one step further: a centralized, managed entity downloads firewall rules to end-hosts (which it identifies with IPSEC certificates in analogy with our use of EIDs to associate policies with hosts). In contrast to the approach described in §6, distributed firewalls do not off-load from clients the job of actually applying rules.

10 Conclusion

The Internet architecture was defined in a context where traffic was benign and addresses plentiful. There were no reasons to interpose functions other than forwarding between endpoints, which became the end-to-end rallying cry for the architecture. Today's Internet is a very different place. There are many reasons why users interpose functions that, in the canonical architecture, either belonged on their host (such as firewalls) or didn't belong at all (such as NATs). The Internet architecture was not designed for—in fact, one might say it was designed against—such interposition of function.

The current incarnations of interposition, middleboxes, are widely derided for their violations of the architecture and the resultant loss of flexibility in the Internet. However, the complexity and risk associated with

being a network host, which used to be minimal, is now daunting even to expert users. We therefore expect outsourcing functionality to become increasingly common.

The architecture presented in this paper, DOA, has a simple goal: to allow the Internet to reap the benefits of network-level middleboxes without their harmful side-effects. It does so not by altering IP, or routers, but by making delegation a basic primitive and introducing a set of globally unique endpoint identifiers.

Acknowledgments

We are grateful to Russ Cox, Bryan Ford, Frans Kaashoek, Karthik Lakshminarayanan, David Mazières, the anonymous reviewers, and our shepherd, Geoff Voelker, for their excellent comments, which substantially improved this paper. Sean Rhea and John Bicket gave useful help with OpenDHT and Click, respectively. We thank Karthik Lakshminarayanan, Sylvia Ratnasamy, and Ion Stoica for useful conversations about naming in the Internet architecture. This work was supported by the NSF under Cooperative Agreement No. ANI-0225660, a Sloan Foundation fellowship, an MIT EECS fellowship, an NSF graduate fellowship, and an NDSEG fellowship.

References

- [1] T. Anderson, T. Roscoe, and D. Wetherall. Preventing Internet denial-of-service with capabilities. In *2nd ACM Workshop on Hot Topics in Networks*, Cambridge, MA, Nov. 2003.
- [2] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking up data in P2P systems. *Communications of the ACM*, 46(2):43–48, Feb. 2003.
- [3] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish. A layered naming architecture for the Internet. In *ACM SIGCOMM*, Portland, OR, Aug. 2004.
- [4] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. *Firmato: A novel firewall management toolkit*. In *IEEE Symposium on Security and Privacy*, May 1999.
- [5] S. M. Bellovin. Distributed firewalls. *login: Magazine, Special Issue on Security*, Nov. 1999.
- [6] M. Borella, D. Grabelsky, J. Lo, and K. Taniguchi. Realm Specific IP: Protocol specification, Oct. 2001. RFC 3103.
- [7] R. Braden, T. Faber, and M. Handley. From protocol stack to protocol heap – role-based architecture. In *1st ACM Workshop on Hot Topics in Networks*, Princeton, NJ, Oct. 2002.
- [8] B. Carpenter and S. Brim. Middleboxes: Taxonomy and issues, Feb 2002. RFC 3234.
- [9] I. Castineyra, N. Chiappa, and M. Steenstrup. The Nimrod routing architecture, Aug 1996. RFC 1992.
- [10] D. Clark, R. Braden, A. Falk, and V. Pingali. FARA: Reorganizing the addressing architecture. In *SIGCOMM FDNA Workshop*, Karlsruhe, Germany, Aug. 2003.
- [11] D. Clark, K. Sollins, J. Wroclawski, and T. Faber. Addressing reality: An architectural response to demands on the evolving Internet. In *ACM SIGCOMM FDNA Workshop*, Karlsruhe, Germany, Aug. 2003.
- [12] D. D. Clark. The design philosophy of the DARPA Internet protocols. In *ACM SIGCOMM*, Stanford, CA, Aug. 1988.
- [13] M. Crawford, A. Mankin, T. Narten, I. J. W. Stewart, and L. Zhang. Separating identifiers and locators in addresses: An analysis of the GSE proposal for IPv6, Oct. 1999. Internet draft

- draft-ietf-ipngwg-esd-analysis-05.txt (Work in progress).
- [14] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. 18th ACM SOSP*, Banff, Canada, Oct. 2001.
 - [15] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6), Dec. 1998. RFC 2460.
 - [16] Dynamic Network Services, Inc., <http://www.dyndns.org>.
 - [17] B. Ford. Unmanaged Internet Protocol: taming the edge network management crisis. In *2nd ACM Workshop on Hot Topics in Networks*, Cambridge, MA, Nov. 2003.
 - [18] B. Ford, P. Srisuresh, and D. Kegel. Peer-to-peer (P2P) communication across middleboxes, Oct. 2003. Internet draft draft-ford-midcom-p2p-01.txt (Work in progress).
 - [19] P. Francis. *Addressing in Internetwork Protocols*. PhD thesis, University College London, UK, 1994.
 - [20] P. Francis and R. Gummadi. IPNL: A NAT-extended Internet architecture. In *ACM SIGCOMM*, San Diego, CA, Aug. 2001.
 - [21] B. Gleeson, A. Lin, J. Heinanen, G. Armitage, and A. Malis. A framework for IP based virtual private networks, Feb. 2000. RFC 2764.
 - [22] M. Gritter and D. R. Cheriton. TRIAD: A new next-generation Internet architecture, <http://www-dsg.stanford.edu/triad/>, July 2000.
 - [23] T. Hain. Architectural implications of NAT, Nov. 2000. RFC 2993.
 - [24] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg. SIP: Session Initiation Protocol, Mar. 1999. RFC 2543.
 - [25] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. Implementing a distributed firewall. In *Computer and Communications Security*, Nov. 2000.
 - [26] J. Jung, Feb. 2004. Personal communication. Data was gathered at MIT using the method described in [27].
 - [27] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. DNS performance and the effectiveness of caching. *IEEE/ACM Trans. on Networking*, 10(5), Oct. 2002.
 - [28] J. Kannan, A. Kubota, K. Lakshminarayanan, I. Stoica, and K. Wehrle. Supporting legacy applications over i3. Technical Report UCB/CSD-04-1342, UC Berkeley, May 2004.
 - [29] B. Karp, S. Ratnasamy, S. Rhea, and S. Shenker. Spurring adoption of DHTs with OpenHash, a public DHT service. In *3rd Intl. Workshop on Peer-to-Peer Systems (IPTPS)*, Feb. 2004.
 - [30] A. D. Keromytis, S. Ioannidis, M. B. Greenwald, and J. M. Smith. The STRONGMAN architecture. In *Third DARPA Information Survivability Conference and Exposition*, Apr. 2003.
 - [31] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Trans. on Computer Systems*, Aug. 2000.
 - [32] H. Krawzyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication, Feb. 1997. RFC 2104.
 - [33] E. Lear and R. Droms. What's in a name: Thoughts from the NSRG, Sept. 2003. draft-irtf-nsrg-report-10, IETF draft (Work in Progress).
 - [34] C. Lynn. Endpoint Identifier Destination Option. Internet Draft, IETF, Nov. 1995. (expired).
 - [35] D. Mazières. A toolkit for user-level file systems. In *Proc. 2001 Usenix Technical Conference*, June 2001.
 - [36] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proc. 17th ACM SOSP*, Kiawah Island, SC, Dec. 1999.
 - [37] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley; 2nd edition, 1996.
 - [38] K. Moore. Things that NATs break, <http://www.cs.utk.edu/~moore/opinions/what-nats-break.html>, as of Oct 2004.
 - [39] R. Moskowitz and P. Nikander. Host identity protocol architecture, Sep 2003. draft-moskowitz-hip-arch-05, IETF draft (Work in Progress).
 - [40] R. Moskowitz, P. Nikander, P. Jokela, and T. Henderson. Host identity protocol, Oct 2003. draft-moskowitz-hip-08, IETF draft (Work in Progress).
 - [41] P. Nikander, J. Ylitalo, and J. Wall. Integrating security, mobility, and multi-homing in a HIP way. In *Network and Distributed Systems Security Symp. (NDSS '03)*, San Diego, CA, Feb 2003.
 - [42] M. O'Dell. 8+8 - An alternate addressing architecture for IPv6, Oct. 1996. Internet draft draft-odell-8+8-00 (Work in progress).
 - [43] M. Ohta. 8+8 addressing for IPv6 end to end multihoming, Jan. 2004. Internet draft draft-ohta-multi6-8plus8-00 (Work in progress).
 - [44] R. Perlman. Understanding ikev2: Tutorial, and rationale for decisions, Feb. 2003. Internet draft draft-ietf-ipsec-ikev2-tutorial-01.txt (Work in progress).
 - [45] V. Ramasubramanian and E. G. Sirer. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In *Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, CA, Mar. 2004.
 - [46] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. Address allocation for private Internets, Feb. 1996. RFC 1918.
 - [47] T. Roscoe, S. Hand, R. Isaacs, R. Mortier, and P. Järdetzy. Predicate routing: Enabling controlled networking. In *1st ACM Workshop on Hot Topics in Networks*, Princeton, NJ, Oct. 2002.
 - [48] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. STUN – simple traversal of user datagram protocol (UDP) through network address translators (NATs), Mar. 2003. RFC 3489.
 - [49] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware*, Heidelberg, Germany, Nov. 2001.
 - [50] J. Saltzer. On the naming and binding of network destinations. In P. Ravasio et al., editor, *Local Computer Networks*, pages 311–317. North-Holland Publishing Company, Amsterdam, 1982. Reprinted as RFC 1498, August 1993.
 - [51] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4), Nov. 1984.
 - [52] J. F. Shoch. Inter-network naming, addressing, and routing. In *17th IEEE Computer Society Conference (COMPCON '78)*, Washington, DC, Sept. 1978.
 - [53] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *Proc. ACM MOBICOM*, 2000.
 - [54] P. Srisuresh and K. Egevang. Traditional IP network address translator (Traditional NAT), Jan. 2001. RFC 3022.
 - [55] P. Srisuresh and M. Holdrege. IP network address translator (NAT) terminology and considerations, Aug. 1999. RFC 2663.
 - [56] P. Srisuresh, J. Kuthan, J. Rosenberg, A. Molitor, and A. Rayhan. Middlebox communication architecture and framework, Aug. 2002. RFC 3303.
 - [57] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. In *ACM SIGCOMM*, Pittsburgh, PA, Aug. 2002.
 - [58] R. P. Swale, P. A. Mart, P. Sijben, S. Brim, and M. Shore. Middlebox communications (MIDCOM) protocol requirements, Aug. 2002. RFC 3304.
 - [59] Symantec Corporation. Norton Personal Firewall, <http://www.symantec.com/sabu/nis/npf/>, as of Oct 2004.
 - [60] VMWare, Inc. <http://www.vmware.com>.
 - [61] M. Walfish, H. Balakrishnan, and S. Shenker. Untangling the Web from DNS. In *Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, CA, Mar. 2004.
 - [62] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, Jan. 2004.

Correlating instrumentation data to system states: A building block for automated diagnosis and control

Ira Cohen
Moises Goldszmidt
Terence Kelly
Julie Symons
HP Labs, Palo Alto, CA

Jeffrey S. Chase
Department of Computer Science
Duke University
Durham, NC
chase@cs.duke.edu

Abstract

This paper studies the use of statistical induction techniques as a basis for automated performance diagnosis and performance management. The goal of the work is to develop and evaluate tools for offline and online analysis of system metrics gathered from instrumentation in Internet server platforms. We use a promising class of probabilistic models (Tree-Augmented Bayesian Networks or TANs) to identify combinations of system-level metrics and threshold values that correlate with high-level performance states—compliance with Service Level Objectives (SLOs) for average-case response time—in a three-tier Web service under a variety of conditions.

Experimental results from a testbed show that TAN models involving small subsets of metrics capture patterns of performance behavior in a way that is accurate and yields insights into the causes of observed performance effects. TANs are extremely efficient to represent and evaluate, and they have interpretability properties that make them excellent candidates for automated diagnosis and control. We explore the use of TAN models for offline forensic diagnosis, and in a limited online setting for performance forecasting with stable workloads.

1 Introduction

Networked computing systems continue to grow in scale and in the complexity of their components and interactions. Today's large-scale network services exhibit complex behaviors stemming from the interaction of workload, software structure, hardware, traffic conditions, and system goals. Pervasive instrumentation and query capabilities are necessary elements of the solution for managing complex systems [32, 23, 33, 14]. There are now many commercial frameworks on the market for coordinated monitoring and control of large-scale systems: tools such as HP's OpenView and IBM's Tivoli aggregate information from a variety of sources and present it

graphically to operators. But it is widely recognized that the complexity of deployed systems surpasses the ability of humans to diagnose and respond to problems rapidly and correctly [17, 26]. Research on automated diagnosis and control—beginning with tools to analyze and interpret instrumentation data—has not kept pace with the demand for practical solutions in the field.

Broadly there are two approaches to building self-managing systems. The most common approach is to incorporate *a priori* models of system structure and behavior, which may be represented quantitatively or as sets of event-condition-action rules. Recent work has explored the uses of such models in automated performance control (e.g., [3, 1, 15]). This approach has several limitations: the models and rule bases are themselves difficult and costly to build, may be incomplete or inaccurate in significant ways, and inevitably become brittle when systems change or encounter unanticipated conditions.

The second approach is to apply statistical learning techniques to induce the models automatically. These approaches assume little or no domain knowledge; they are therefore generic and have potential to apply to a wide range of systems and to adapt to changes in the system and its environment. For example, there has been much recent progress on the use of statistical analysis tools to infer component relationships from histories of interaction patterns (e.g., from packet traces) [9, 2, 4, 10]. But it is still an open problem to identify techniques that are powerful enough to induce effective models, and that are sufficiently efficient, accurate, and robust to deploy in practice.

The goal of our work is to automate analysis of instrumentation data from network services in order to forecast, diagnose, and repair failure conditions. This paper studies the effectiveness and practicality of *Tree-Augmented Naive* Bayesian networks [18], or TANs, as a basis for performance diagnosis and forecasting from system-level instrumentation in a three-tier network service. TANs comprise a subclass of Bayesian networks,

recently of interest to the systems community as potential elements of an Internet “Knowledge Plane” [11]. TANs are less powerful than generalized Bayesian networks (see Section 3), but they are simple, compact and efficient. TANs have been shown to be promising in diverse contexts including financial modeling, medical diagnosis, text classification, and spam filtering, but we are not aware of any previous study of TANs in the context of computer systems.

To explore TANs as a basis for self-managing systems, we analyzed data from 124 metrics gathered from a three-tier e-commerce site under synthetic load. The induced TAN models select combinations of metrics and threshold values that correlate with high-level performance states—compliance with Service Level Objectives (SLO) for average response time—under a variety of conditions. The experiments support the following conclusions:

- Combinations of metrics are significantly more predictive of SLO violations than individual metrics. Moreover, different combinations of metrics and thresholds are selected under different conditions. This implies that even this relatively simple problem is too complex for simple “rules of thumb” (e.g., just monitor CPU utilization).
- Small numbers of metrics (typically 3–8) are sufficient to predict SLO violations accurately. In most cases the selected metrics yield insight into the cause of the problem and its location within the system. This property of *interpretability* is a key advantage of TAN models (Section 3.3). While we do not claim to solve the problem of root cause analysis, our results suggest that TANs have excellent potential as a basis for diagnosis and control. For example, we may statically associate metrics with control variables (actuators) to restore the system to a desired operating range.
- Although multiple metrics are involved, the relationships among these metrics are relatively simple in this context. Thus TAN models are highly accurate: in typical cases, the models yield a *balanced accuracy* of 90%–95% (see Section 2.1).
- The TAN models are extremely efficient to represent and evaluate. Model induction is efficient enough to adapt to changes in workload and system structure by continuously inducing new models.

Of the known statistical learning techniques, the TAN structure and algorithms are among the most promising for deployment in real systems. They are based on sound and well-developed theory, they are computationally efficient and robust, they require no expertise to use, and

they are readily available in open-source implementations [24, 34, 5]. While other approaches may prove to yield comparable accuracy and/or efficiency, Bayesian networks and TANs in particular have important practical advantages: they are interpretable and they can incorporate expert knowledge and constraints. Although our primary emphasis is on *diagnosing* performance problems after they have occurred, we illustrate the versatility of TANs by using them to *forecast* problems. We emphasize that our methods discover correlations rather than causal connections, and the results do not yet show that a robust “closed loop” diagnosis is practical at this stage. Even so, the technique can sift through a large amount of instrumentation data rapidly and focus the attention of a human analyst on the small set of metrics most relevant to the conditions of interest.

This paper is organized as follows: Section 2 defines the problem and gives an overview of our approach. Section 3 gives more detail on TANs and the algorithms to induce them, and outlines the rationale for selecting this technique for computer systems diagnosis and control. Section 4 describes the experimental methodology and Section 5 presents results. Section 6 presents additional results from a second testbed to confirm the diagnostic power of TANs. Section 7 discusses related work, and Section 8 concludes.

2 Overview

Figure 1 depicts the experimental environment. The system under test is a three-tier Web service: the Web server (Apache), application middleware server (BEA WebLogic), and database server (Oracle) run on three different servers instrumented with HP OpenView to collect a set of system metrics. A load generator (`httperf` [28]) offers load to the service over a sequence of execution intervals. An SLO indicator processes the Apache logs to determine SLO compliance over each interval, based on the average server response time for requests in the interval.

This paper focuses on the problem of constructing an analysis engine to process the metrics and indicator values. The goal of the analysis is to induce a *classifier*, a function that predicts whether the system is or will be in compliance over some interval, based on the values of the metrics collected. If the classifier is interpretable, then it may also be useful for diagnostic forensics or control. One advantage of our approach is that it identifies sets of metrics and threshold values that correlate with SLO violations. Since specific metrics are associated with specific components, resources, processes, and events within the system, the classifier indirectly identifies the system elements that are most likely to be involved with the failure or violation. Even so, the analysis

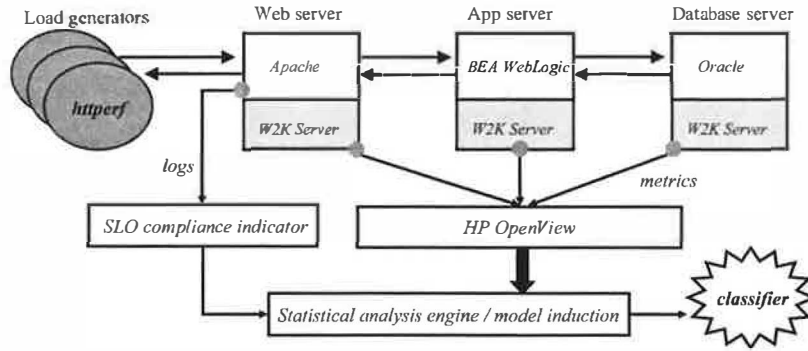


Figure 1: This study explores the use of TAN classifiers to diagnose a common type of network service: a three-tier Web application with a Java middleware component, backed by a database.

process is general because it uses no *a priori* knowledge of the system's structure or function.

In this study we limit our attention to system-level metrics gathered from a standard operating system (Windows 2000 Server) on each of the servers. Of course, the analysis engine may be more effective if it considers application-level metrics; on the other hand, analysis and control using system-level metrics can generalize to any application. Table 1 lists some specific system-level metrics that are often correlated with SLO violations.

2.1 Formalizing the Problem

This problem is a pattern classification problem in supervised learning. Let S_t denote the state of the SLO at time t . In this case, S can take one of two states from the set $\{compliance, violation\}$: let $\{0, 1\}$ or $\{s^+, s^-\}$ denote these states. Let \vec{M}_t denote a vector of values for n collected metrics $[m_0, \dots, m_n]$ at time t (we will omit the subindex t when the context is clear). The pattern classification problem is to induce or learn a classifier function \mathcal{F} mapping the universe of possible values for \vec{M}_t to the range of system states S [16, 7].

The input to this analysis is a training data set. In this case, the training set is a log of observations of the form $\langle \vec{M}_t, S_t \rangle$ from the system in operation. The learning is supervised in that the SLO compliance indicator identifies the value of S_t corresponding to each observed \vec{M}_t in the training set, providing preclassified instances for the analysis to learn from.

We emphasize four premises that are implicit in this problem statement. First, it is not necessary to predict system behavior, but only to identify system states that correlate with particular failure events (e.g., SLO violations). Second, the events of interest are defined and identified externally by some failure detector. For example, in this case it is not necessary to predict system performance, but only to classify system states that com-

ply with an SLO as specified by an external indicator. Third, there are patterns among the collected metrics that correlate with SLO compliance; in this case, the metrics must be well-chosen to capture system states relating to the behavior of interest. Finally, the analysis must observe a statistically significant sample of event instances in the training set to correlate states with the observed metrics. Our approach is based on classification rather than anomaly detection: it trains the models with observations of SLO violations as well as normal behavior. The resulting models are useful to predict and diagnose performance problems.

The key measure of success is the accuracy of the resulting classifier \mathcal{F} . A common metric is the *classification accuracy*, which in this case is defined as the probability that \mathcal{F} correctly identifies the SLO state S_t associated with any \vec{M}_t . This measure can be misleading when violations are uncommon: for example, if 10% of the intervals violate the SLO, a trivial classifier that always guesses compliance yields a classification accuracy of 90%. Instead, our figure of merit is *balanced accuracy* (BA), which averages the probability of correctly identifying compliance with the probability of detecting a violation. Formally:

$$BA = \frac{P(s^- = \mathcal{F}(\vec{M})|s^-) + P(s^+ = \mathcal{F}(\vec{M})|s^+)}{2} \quad (1)$$

To achieve the maximal BA of 100%, \mathcal{F} must perfectly classify both SLO violation and SLO compliance. The trivial classifier in the example above has a BA of only 50%. In some cases we can gain more insight into the behavior of a classifier by considering the false positive rate and false negative rate separately.

2.2 Inducing Classifier Models

There are many techniques for pattern classification in the literature (e.g., [7, 30]). Our approach first induces

Metric	Description
mean_AS_CPU_1_USERTIME	CPU time spent in user mode on the application server.
var_AS_CPU_1_USERTIME	Variance of user CPU time on the application server.
mean_AS_DISK_1_PHYSREAD	Number of physical disk reads for disk 1 on the application server, includes file system reads, raw I/O and virtual memory I/O.
mean_AS_DISK_1_BUSYTIME	Time in seconds that disk 1 was busy with pending I/O on the application server.
var_AS_DISK_1_BUSYTIME	Variance of time that disk 1 was busy with pending I/O on the application server.
mean_DB_DISK_1_PHYSWRITEBYTE	Number of kilobytes written to disk 1 on the database server, includes file system reads, raw I/O and virtual memory I/O.
var_DB_GBL_SWAPSPACEUSED	Variance of swap space allocated on the database server.
var_DB_NETIF_2_INPACKET	Variance of the number of successful (no errors or collisions) physical packets received through network interface #2 on the database server.
mean_DB_GBL_SWAPSPACEUSED	Amount of swap space, in MB, allocated on the database server.
mean_DB_GBL_RUNQUEUE	Approximate average queue length for CPU on the database server.
var_DB_NETIF_2_INBYTE	Variance of the number of KBs received from the network via network interface #2 on the database server. Only bytes in packets that carry data are included.
var_DB_DISK_1_PHYSREAD	Variance of physical disk reads for disk 1 on the database server.
var_AS_GBL_MEMUTIL	Variance of the percentage of physical memory in use on the application server, including system memory (occupied by the kernel), buffer cache, and user memory.
numReqs	Number of requests the system has served.
var_DB_DISK_1_PHYSWRITE	Variance of the number of writes to disk 1 on the database server.
var_DB_NETIF_2_OUTPACKET	Variance of the number of successful (no errors or collisions) physical packets sent through network interface #2 on the database server.

Table 1: A sampling of system-level metrics that are often correlated with SLO violations in our experiments, as named by HP OpenView. “AS” refers to metrics measured on the application server; “DB” refers to metrics measured on the database server.

a *model* of the relationship between \vec{M} and S , and then uses the model to decide whether any given set of metric values \vec{M} is more likely to correlate with an SLO violation or compliance. In our case, the model represents the conditional distribution $P(S|\vec{M})$ —the distribution of probabilities for the system state given the observed values of the metrics. The classifier then uses this distribution to evaluate whether $P(s^+|\vec{M}) > P(s^-|\vec{M})$.

Thus, we transform the problem of pattern classification to one of statistical fitting of a probabilistic model. The key to this approach is to devise a way to represent the probability distribution that is compact, accurate, and efficient to process. Our approach represents the distribution as a form of Bayesian network (Section 3).

An important strength of this approach is that one can interrogate the model to identify specific metrics that affect the classifier’s choice for any given \vec{M} . This *interpretability* property makes Bayesian networks attractive for diagnosis and control, relative to competing alternatives such as neural networks and support vector machines [13]. One other alternative, decision trees [30], can be interpreted as a set of if-then rules on the metrics and their values. Bayesian networks have an additional advantage of *modifiability*: they can incorporate expert knowledge or constraints into the model efficiently. For example, a user can specify a subset of metrics or correlations to include in the model, as discussed below. Sec-

tion 3.3 outlines the formal basis for these properties.

The key challenge for our approach is that it is intractable to induce the optimal Bayesian network classifier. Heuristics may guide the search for a good classifier, but there is also a risk that a generalized Bayesian network may overfit data from the finite and possibly noisy training set, compromising accuracy. Instead, we restrict the form of the Bayesian network to a TAN (Section 3) and select the optimal TAN classifier over a heuristically selected subset of the metrics. This approach is based on the premise (which we have validated empirically in our domain) that a relatively small subset of metrics and threshold values is sufficient to approximate the distribution accurately in a TAN encoding relatively simple dependence relationships among the metrics. Although the effectiveness of TANs is sensitive to the domain, TANs have been shown to outperform generalized Bayesian networks and other alternatives in both cost and accuracy for classification tasks in a variety of contexts [18]. This paper evaluates the efficiency and accuracy of the TAN algorithm in the context of SLO maintenance for a three-tier Web service, and investigates the nature of the induced models.

2.3 Using Classifier Models

Before explaining the approach in detail, we first consider its potential impact in practice. We are interested in using classifiers to diagnose a failure or violation condition, and ultimately to repair it.

The technique can be used for diagnostic forensics as follows. Suppose a developer or operator wishes to gain insight into a system's behavior during a specific execution period for which metrics were collected. Running the algorithm yields a classifier for any event—such as a failure condition or SLO threshold violation—that occurs a sufficient number of times to induce a model (see Section 3). In the general case, the event may be defined by any user-specified predicate (indicator function) over the metrics. The resulting model gives a list of metrics and ranges of values that correlate with the event, selected from the metrics that do not appear in the definition of the predicate.

The user may also “seed” the models by preselecting a set of metrics that must appear in the models, and the value ranges for those metrics. This causes the algorithm to determine the degree to which those metrics and value ranges correlate with the event, and to identify additional metrics that are maximally correlated subject to the condition that the values of the specified metrics are within their specified ranges. For example, a user can ask a question of the form: “what percentage of SLO violations occur during intervals when the network traffic between the application server and the database server is high, and what other metrics and values are most predictive of SLO violations during those intervals?”

The models also have potential to be useful for online forecasting of failures or SLO violations. For example, Section 5 shows that it is possible to induce models that predict SLO violations in the near future, when the characteristics of the workload and system are stable. An automated controller may invoke such a classifier directly to identify impending violations and respond to them, e.g., by shedding load or adding resources.

Because the models are cheap to induce, the system may refresh them periodically to track changes in the workload characteristics and their interaction with the system structure. In more dynamic cases, it is possible to maintain multiple models in parallel and select the best model for any given period. The selection criteria may be based on recent accuracy scores, known cyclic behavior, or other recognizable attributes.

3 Approach

This section gives more detail on the TAN representation and algorithm, and discusses the advantages of this approach relative to its alternatives.

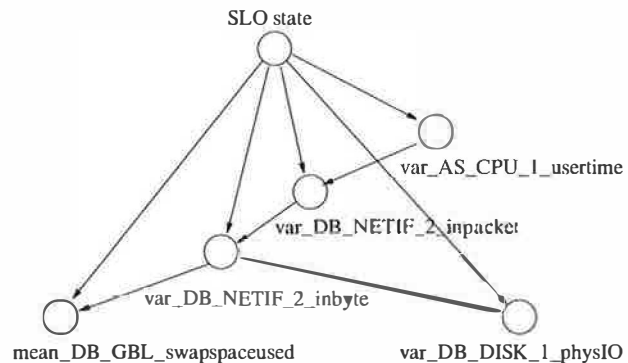


Figure 2: Example TAN to fit SLO violations in a three-tier Web service. Table 1 defines the metrics.

As stated in the previous section, we use TANs to obtain a compact, efficient representation of the model underlying the classifier. The model approximates a probability distribution $P(S|\vec{M})$, which gives the probability that the system is in any given state S for any given vector of observed metrics \vec{M} . Inducing a model of this form reduces to fitting the distribution $P(\vec{M}|S)$ —the probability of observing a given vector \vec{M} of metric values when the system is in a given state S . Multidimensional problems of this form are subject to challenges of robustness and overfitting, and require a large number of data samples [16, 21]. We can simplify the problem by making some assumptions about the structure of the distribution P . TANs comprise a subclass of Bayesian networks [29], which offer a well-developed mathematical language to represent structure in probability distributions.

3.1 Bayesian networks and TANs

A Bayesian network is an annotated directed acyclic graph encoding a joint probability distribution. The vertices in the graph represent the random variables of interest in the domain to be modeled, and the edges represent direct influences of one variable on another. In our case, each system-level metric m_i is a random variable represented in the graph. Each vertex in the network encodes a probability distribution on the values that the random variable can take, given the state of its predecessors. This representation encodes a set of (probabilistic) independence statements of the form: each random variable is independent of its non-descendants, given that the state of its parents is known. There is a set of well-understood algorithms and methods to induce Bayesian network models statistically from data [22], and these are available in open-source software [24, 34, 5].

In a *naïve* Bayesian network, the state variable S is the only parent of all other vertices. Thus a naïve Bayesian network assumes that all the metrics are fully indepen-

dent given S . A tree-augmented naive Bayesian network (TAN) extends this structure to consider relationships among the metrics themselves, with the constraint that each metric m_i has at most one parent m_{p_i} in the network other than S . Thus a TAN imposes a tree-structured dependence graph on a naive Bayesian network; this structure is a *Markov tree*. The TAN for a set of observations and metrics is defined as the Markov tree that is *optimal* in the sense that it has the highest probability of having generated the observed data [18].

Figure 2 illustrates a TAN obtained for one of our experiments (see the STEP workload in Section 4.1.2). This model has a balanced accuracy (BA) score of 94% for the system, workload, and SLO in that experiment. The metrics selected are the variance of the CPU user time at the application server, network traffic (packets and bytes) from that server to the database tier, and the swap space and disk activity at the database. The tree structure captures the following assertions: (1) given the network traffic between the tiers, the CPU activity in the application server is irrelevant to the swap space and disk activity at the database tier; (2) the network traffic is correlated with CPU activity, i.e., common increases in the values of those metrics are not anomalous.

Our TAN models approximate the probability distribution of values for each metric (given the value of its predecessor) as a conditional Gaussian distribution. This method is efficient and avoids problems of discretization. The experimental results show that it has acceptable accuracy and is effective in capturing the abnormal metric values associated with each performance state. Other representations may be used with the TAN technique.

3.2 Selecting a TAN model

Given a basic understanding of the classification approach and the models, we now outline the methods and algorithms used to select the TAN model for the classifier (derived from [18]). The goal is to select a subset \vec{M}^* of \vec{M} whose TAN yields the most accurate classifier, i.e., \vec{M}^* includes the metrics from \vec{M} that correlate most strongly with SLO violations observed in the data. Let k be the size of the subset \vec{M}^* . The problem of selecting the best k metrics for \vec{M}^* is known as *feature selection*. Most solutions use some form of heuristic search given the combinatorial explosion of the search space in the number of metrics in \vec{M} . We use a greedy strategy: at each step select the metric that is not already in the vector \vec{M}^* , and that yields maximum improvement in accuracy (BA) of the resulting TAN over the sample data. To do this, the algorithm computes the optimal Markov tree for each candidate metric, then selects the metric whose tree yields the highest BA score against the observed data. The cost is $O(kn)$ times the cost to induce and evaluate

the Markov tree, where n is the number of metrics. The algorithm to find the optimal Markov tree computes a minimum spanning tree over the metrics in \vec{M}^* .

From Eq. 1 it is clear that to compute a candidate's BA score we must estimate the probability of false positives and false negatives for the resulting model. The algorithm must approximate the real BA score from a finite set of samples. To ensure the robustness of this score against variability on the unobserved cases in the data, the following procedure called *ten-fold cross validation* is used [21]. Randomly divide the data into two sets, a training set and a testing set. Then, induce the model with the training set, and compute its score with the testing set. Compute the final score as the average score over ten trials. This reduces any biases or overfitting effects resulting from a finite data set.

Given a data set with N samples of the n metrics, the overall algorithm is dominated by $O(n^2 \cdot N)$ for small k , when all N samples are used to induce and test the candidates. Most of our experiments train models for 31 SLO definitions on stored instrumentation datasets with $n = 124$ and $N = 2400$. Our Matlab implementation processes each dataset in about ten minutes on a 1.8 GHz Pentium 4 (~ 20 seconds per SLO). Each run induces about 40,000 candidate models, for a rough average of 15 ms per model. Once the model is selected, evaluating it to classify a new interval sample takes 1-10 ms. These operations are cheap enough to train models online as needed and even to maintain and evaluate multiple models in parallel.

3.3 Interpretability and Modifiability

In addition to their efficiency in representation and inference, TANs (and Bayesian networks in general) present two key practical advantages: *interpretability* and *modifiability*. These properties are especially important in the context of diagnosis and control.

The influence of each metric on the violation of an SLO can be quantified in a sound probabilistic model. Mathematically, we arrive at the following functional form for the classifier as a sum of terms, each involving the probability that the value of some metric m_i occurs in each state given the value of its predecessor m_{p_i} :

$$\sum_i \log \left[\frac{P(m_i | m_{p_i}, s^-)}{P(m_i | m_{p_i}, s^+)} \right] + \log \frac{P(s^-)}{P(s^+)} > 0 \quad (2)$$

Each metric is essentially subjected to a likelihood test comparing the probability that the observed value occurs during compliance to the probability that the value occurs during violation. A sum value greater than zero indicates a violation. This analysis catalogs each type of SLO violation according to the metrics and values that correlate with observed instances. Furthermore, the

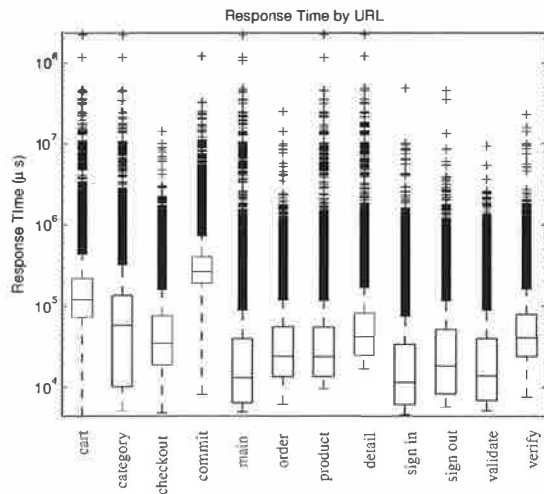


Figure 3: Observed distributions of response time for PetStore operations. Each box marks the quartiles of the distribution; the horizontal line inside each box is the median. Outliers are shown as crosses outside each box.

strength of each metric's influence on the classifier's choice is given from the probability of its value occurring in the different states.

This structure gives insight into the causes of the violation or even how to repair it. For example, if violation of a temperature threshold is highly correlated with an open window, then one potential solution may be to close the window. Of course, any correlation is merely "circumstantial evidence" rather than proof of causality; much of the value of the analysis is to "exonerate" the metrics that are not correlated with the failure rather than to "convict the guilty".

Because these models are interpretable and have clear semantics in terms of probability distributions, we can enhance and complement the information induced directly from data with expert knowledge of the domain or system under study [22]. This knowledge can take the form of explicit lists of metrics to be included in the model, information about correlations and dependencies among the metrics, or prior probability distributions. Blake & Breese [8] give examples, including an early use of Bayesian networks to discover bottlenecks in the Windows operating system. Sullivan [31] applies this approach to tune database parameters.

4 Methodology

We considered a variety of approaches to empirical evaluation before eventually settling on the testbed environment and workloads described in this section. We rejected the use of standard synthetic benchmarks, e.g., TPC-W, because they typically ramp up load to a stable

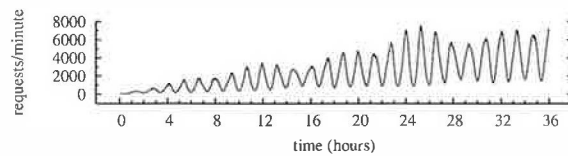


Figure 4: Requests per minute in RAMP workload.

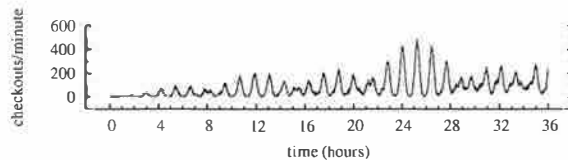


Figure 5: Purchases per minute in RAMP workload.

plateau in order to determine peak throughput subject to a constraint on mean response time. Such workloads are not sufficiently rich to produce the wide range of system conditions that might occur in practice. Traces collected in real production environments are richer, but production systems rarely permit the controlled experiments necessary to validate our methods. For these reasons we constructed a testbed with a standard three-tiered Web server application—the well-known Java PetStore—and subjected it to synthetic stress workloads designed to expose the strengths and limitations of our approach.

The Web, application, and database servers were hosted on separate HP NetServer LPr systems configured with a Pentium II 500 MHz processor, 512 MB of RAM, one 9 GB disk drive and two 100 Mbps network cards. The application and database servers run Windows 2000 Server SP4. We used two different configurations of the Web server: Apache Version 2.0.48 with a BEA WebLogic plug-in on either Windows 2000 Server SP4 or RedHat Linux 7.2. The application server runs BEA WebLogic 7.0 SP4 over Java 2 SDK Version 1.3.1 (08) from Sun. The database client and server are Oracle 9iR2. The testbed has a switched 100 Mbps full-duplex network.

The experiments use a version of the Java PetStore obtained from the Middleware Company in October 2002. We tuned the deployment descriptors, `config.xml`, and `startWebLogic.cmd` in order to scale to the transaction volumes reported in the results. In particular, we modified several of the EJB deployment descriptors to increase the values for `max-beans-in-cache`, `max-beans-in-free-pool`, `initial-bean-in-free-pool`, and some of the timeout values. The concurrency-strategy for two of the beans in the Customer and Inventory deployment descriptors was changed to "Database". Other changes include increasing the execute thread count to 30, increasing the initial and maximum ca-

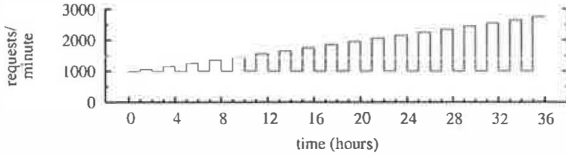


Figure 6: Requests per minute in STEP workload.

capacities for the JDBC Connection pool, increasing the PreparedStatementCacheSize, and increasing the JVM's maximum heap size. The net effect of these changes was to increase the maximum number of concurrent sessions from 24 to over 100.

Each server is instrumented using the HP OpenView Operations Embedded Performance Agent, a component of the OpenView Operations Agent, Release 7.2. We configured the agent to sample and collect values for 124 system-level metrics (e.g., including the metrics listed in Table 1) at 15-second intervals.

4.1 Workloads

We designed the workloads to exercise our model-induction methodology by providing it with a wide range of (\vec{M}, \vec{P}) pairs, where \vec{M} represents a sample of values for the system metrics and \vec{P} represents a vector of application-level performance measurements (e.g., response time & throughput). Of course, we cannot directly control either \vec{M} or \vec{P} ; we control only the exogenous workload submitted to the system under test. We vary several characteristics of the workload, including

1. aggregate request rate,
2. number of concurrent client connections, and
3. fraction of requests that are database-intensive (e.g., checkout) vs. app-server-intensive (e.g., browsing).

Figure 3 presents box plots depicting the response time distributions of the twelve main request classes in our PetStore testbed. Response times differ significantly for different types of requests, hence the request mix is quite versatile in its effect on the system.

We mimic key aspects of real-world workload, e.g., varying burstiness at fine time scales and periodicity on longer time scales. However, each experiment runs in 1–2 days, so the periods of workload variation are shorter than in the wild. We wrote simple scripts to generate session files for the `httperf` workload generator [28], which allows us to vary the client think time and the arrival rate of new client sessions.

4.1.1 RAMP: Increasing Concurrency

In this experiment we gradually increase the number of concurrent client sessions. We add an emulated client

```

1 For each Experiment
2   For SLO threshold = 60, ..., 90 percentile
     of average response time
3     Identify intervals violating SLO.
4     Select k metrics using greedy search.
5     Induce TAN model with top k metrics.
6     Evaluate with 10-fold cross validation
       for balanced accuracy, false alarm
       and detection rates.
7   Evaluate using only Application server
     CPU usertime metric ('CPU').
8   if (SLO threshold == 60 percentile)
9     Save model as 'MOD'.
10  else
11    Evaluate MOD on current SLO.
12  Record metric attribution of current
     TAN for each interval violating SLO.

```

Table 2: The testing procedure.

every 20 minutes up to a limit of 100 total sessions, and terminate the test after 36 hours. *Individual* client request streams are constructed so that the *aggregate* request stream resembles a sinusoid overlaid upon a ramp; this effect is depicted in Figure 4, which shows the ideal throughput of the system under test. The *ideal* throughput occurs *if all requests are served instantaneously*. Because `httperf` uses a closed client loop with think time, the actual rate depends on response time.

Each client session follows a simple pattern: go to main page, sign in, browse products, add some products to shopping cart, check out, repeat. Two parameters indirectly define the number of operations within a session. One is the probability that an item is added to the shopping cart given that it has just been browsed. The other is the probability of proceeding to the checkout given that an item has just been added to the cart. These probabilities vary sinusoidally between 0.42 and 0.7 with periods of 67 and 73 minutes, respectively. The net effect is the ideal time-varying checkout rate shown in Figure 5.

4.1.2 STEP: Background + Step Function

This 36-hour run has two workload components. The first `httperf` creates a steady background traffic of 1000 requests per minute generated by 20 clients. The second is an on/off workload consisting of hour-long bursts with one hour between bursts. Successive bursts involve 5, 10, 15, etc. client sessions, each generating 50 requests per minute. Figure 6 summarizes the ideal request rate for this pattern, omitting fluctuations at fine time scales.

The intent of this workload is to mimic sudden, sustained bursts of increasingly intense workload against a backdrop of moderate activity. Each “step” in the workload produces a different plateau of workload level, as well as transients during the beginning and end of each

experiment	SLO thresh (msec)	Avg # Metrics	TAN BA	MOD BA	CPU BA	TAN FA	MOD FA	CPU FA	TAN Det	MOD Det	CPU Det
RAMP	62 – 627	3	94 ±2.4	84 ±5	90 ±8	6.4 ±2	29 ±11	8.5 ±4	93 ±2	98 ±0.3	88.7 ±19
STEP	111 – 541	8	92.7 ±2	89.9 ±2.6	56 ±8.8	6.6 ±2.9	16 ±8.2	13 ±16	91.9 ±4.5	96 ±4.2	27 ±34
BUGGY	214 – 627	4	87.3 ±3.3	86.4 ±3.2	63.4 ±12.1	16.9 ±6.8	21.0 ±7.2	14.9 ±13.3	91.6 ±3.1	94.2 ±1.02	41.7 ±37.6

Table 3: Summary of accuracy results. BA is balanced accuracy, FA is false alarm and Det is detection.

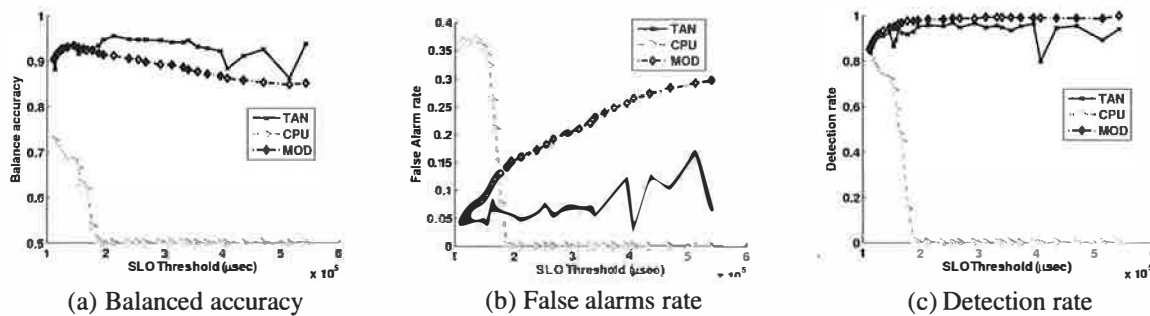


Figure 7: Accuracy results for STEP as a function of SLO threshold. The TAN trained for a given workload and SLO balances the rates of detection and false alarms for the highest balanced accuracy (BA).

step as the system adapts to the change.

4.1.3 BUGGY: Numerous Errors

BUGGY was a five-hour run with 25 client sessions. Aggregate request rate ramped from 1 request/sec to 50 requests/sec during the course of the experiment, with sinusoidal variation of period 30 minutes overlaid upon the ramp. The probability of add-to-cart following browsing an item and the probability of checkout following add-to-cart vary sinusoidally between 0.1 and 1 with periods of 25 and 37 minutes, respectively. This run occurred before the Petstore deployment was sufficiently tuned as described previously. The J2EE component generated numerous Java exceptions, hence the title “BUGGY.”

5 Experimental Results

This section evaluates our approach using the system and workloads described in Section 4. In these experiments we varied the SLO threshold to explore the effect on the induced models, and to evaluate accuracy of the models under varying conditions. For each workload, we trained and evaluated a TAN classifier for each of 31 different SLO definitions, given by varying the threshold on the average response time such that the percentage of intervals violating the SLO varies from 40% to 10% in increments of 1%. As a baseline, we also evaluated the accuracy of the 60-percentile SLO classifier (MOD) and a

simple “rule of thumb” classifier using application server CPU utilization as the sole indicator metric. Table 2 summarizes the testing procedure.

Table 3 summarizes the average accuracy of all models across all SLO thresholds for each workload. Figure 7 plots the results for all 31 SLO definitions for STEP. We make several observations from the results:

1. Overall balanced accuracy of the TAN model is high, ranging from 87%-94%. In a breakdown of false alarms to detection rates we see that detection rates are higher than 90% for all experiments, with false alarms at about 6% for two experiments and 17% for BUGGY.
2. A single metric alone (CPU in this case) is not sufficient to capture the patterns of SLO violations. While CPU has a BA score of 90 for RAMP, it does very poorly for the other two workloads. To illustrate, Figure 8 plots average response time for each interval in the STEP run as a function of its average CPU utilization. The plot shows that while CPU usage correlates with average latency when latency is low, the correlation is not apparent for intervals with high average latency. Indeed, Figure 7 shows that the low BA score stems from a low detection rate for the less stringent SLO thresholds.
3. A small number of metrics is sufficient to capture the patterns of SLO violations. The number of metrics in the TAN models ranges from 3 to 8.

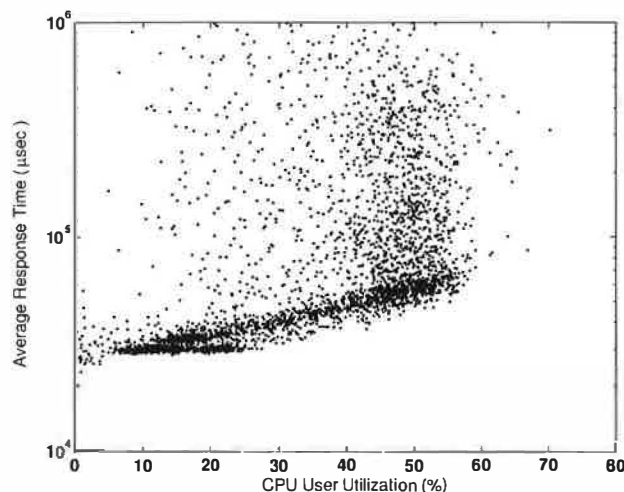


Figure 8: Average response time as a function of the application server CPU user utilization for STEP.

4. The models are sensitive to the workload and SLO definition. For example, the accuracy of MOD (the TAN model for the most stringent SLO) always has a high detection rate on the less stringent SLOs (as expected), but generates false alarms at an increasing rate as the SLO threshold increases.

Determining the number of metrics. To illustrate the role of multiple metrics in accurate TAN models, Figure 9 shows the top three metrics (in order) as a function of average response time for the STEP workload with SLO threshold of 313 msec (20% instances of SLO violations). The top metric alone yields a BA score of 84%, which improves to 88% with the second metric. However, by itself, the second metric is not discriminative; in fact, the second metric alone yields a BA of just 69%. The TAN combines these metrics for higher accuracy by representing their relationships. Adding two more metrics increases the BA score to 93.6%.

Interaction between metrics and values. The metrics selected for a TAN model may have complex relationships and threshold values. The combined model defines decision boundaries that classify the SLO state (violation/no violation) of an interval by relating the recorded values of the metrics during the interval. Figure 10 depicts the decision boundary learned by a TAN model for its top two metrics. The figure also shows the best decision boundary when these metrics are used in isolation. We see that the top metric is a fairly good predictor of violations, while the second metric alone is poor. However, the decision boundary of the model with both metrics takes advantage of the strength of both metrics and “carves out” a region of value combinations that correlate with SLO violations.

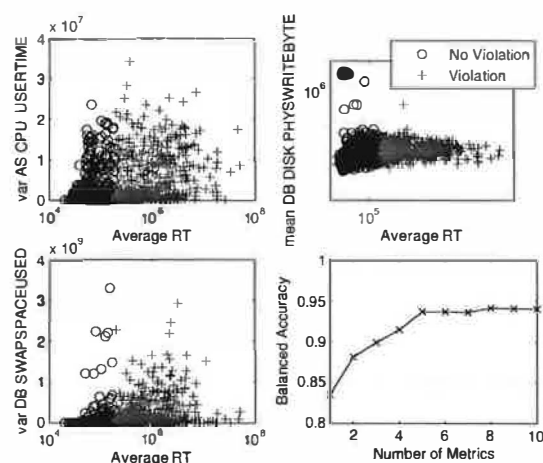


Figure 9: Plots of the top three metrics selected for a TAN model for the STEP workload. Modeling the correlations of five metrics yields a BA of 93.6%, a significant improvement over any of the metrics in isolation.

Adaptation. Additional analysis shows that the models must adapt to capture the patterns of SLO violation with different response time thresholds. For example, Figure 7 shows that the metrics selected for MOD have a high detection rate across all SLO thresholds, but the increasing false alarm rate indicates that it may be necessary to adjust their threshold values and decision boundaries. However, it is often more effective to adapt the metrics as conditions change.

To illustrate, Table 4 lists the metrics selected for at least six of the SLO definitions in either the RAMP or STEP experiments. The most commonly chosen metrics differ significantly across the workloads, which stress the testbed in different ways. For RAMP, CPU usertime and disk reads on the application server are the most common, while swap space and I/O traffic at the database tier are most highly correlated with SLO violations for STEP. A third and entirely different set of metrics is selected for BUGGY: all of the chosen metrics are related to disk usage on the application server. Since the instrumentation records only system-level metrics, disk traffic is most highly correlated with the server errors occurring during the experiment, which are logged to disk.

Metric “Attribution”. The TAN models identify the metrics that are most relevant—alone or in combination—to SLO violations, which is a key step toward a root-cause analysis. Figure 11 demonstrates metric attribution for RAMP with SLO threshold set at 100msec (20% of the intervals are in violation). The model includes two metrics drawn from the application server: CPU user time and disk reads. We see that most SLO violations are attributed to high CPU utilization,

Metric/exper #	RAMP	STEP
mean_AS_CPU_1_USERTIME	27	7
mean_AS_DISK_1_PHYSREAD	14	0
mean_AS_DISK_1_BUSYTIME	6	0
var_AS_DISK_1_BUSYTIME	6	0
mean_DB_DISK_1_PHYSWRITEBYTE	1	22
var_DB_GBL_SWAPSPACEUSED	0	21
var_DB_NETIF_2_INPACKET	2	21
mean_DB_GBL_SWAPSPACEUSED	0	14
mean_DB_GBL_RUNQUEUE	0	13
var_AS_CPU_1_USERTIME	0	12
var_DB_NETIF_2_INBYTE	0	10
var_DB_DISK_1_PHYSREAD	0	9
var_AS_GBL_MEMUTIL	0	8
numReqs	0	7
var_DB_DISK_1_PHYSWRITE	0	6
var_DB_NETIF_2_OUTPACKET	5	6

Table 4: Counts of the number of times the most commonly selected metrics appear in TAN models for SLO violations in the RAMP and STEP workloads. See Table 1 for a definition of the metrics.

while some instances are explained by the combination of CPU and disk traffic, or by disk traffic alone. For this experiment, violations occurring as sudden short spikes in average response time were explained solely by disk traffic, while violations occurring during more sustained load surges were attributed mostly to high CPU utilization, or to a combination of both metrics.

Forecasting. Finally, we consider the accuracy of TAN models in forecasting SLO violations. Table 5 shows the accuracy of the models for forecasting SLO violations three sampling intervals in advance (sampling interval is 5 minutes for STEP and 1 minute for the others). The models are less accurate for forecasting, as expected, but their BA scores are still 80% or higher. Forecasting accuracy is sensitive to workload: the RAMP workload changes slowly, so predictions are more accurate than for the bursty STEP workload. Interestingly, the metrics most useful for forecasting are not always the same ones selected for diagnosis: a metric that correlates with violations as they occur is not necessarily a good predictor of future violations.

exper.	TAN BA	TAN FA	TAN Det
RAMP	91.8 ±1.2	9.1 ±3	93 ±3.1
STEP	79.7 ±4.6	24 ±5.5	83 ±7.4
BUGGY	79.7 ±4.6	24 ±7.4	83.4 ±5.6

Table 5: Summary of forecasting results. BA is balanced accuracy, FA is false alarm and Det is detection.

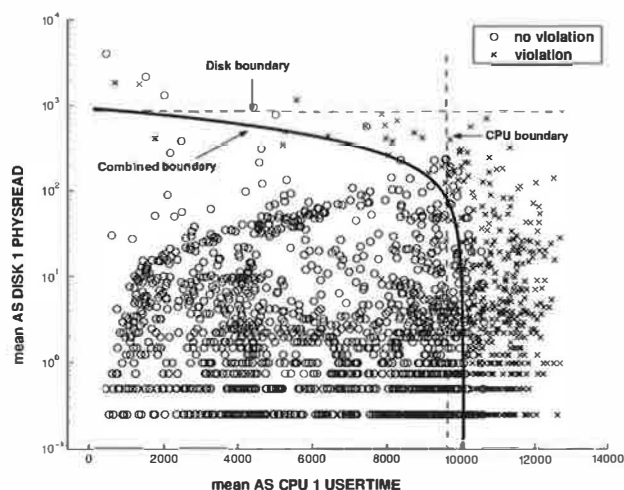


Figure 10: Decision boundary of a TAN model for the values of its top two metrics. Horizontal and vertical lines show decision boundaries for the individual metrics in isolation. Combining the metrics yields higher accuracy than either metric in isolation.

6 Validation with Independent Testbed

To further validate our methods, we analyzed data collected by a group of HP's OpenView developers on a different Web service testbed. The important characteristic of these tests is that they induce performance behaviors and SLO violations with a second application that contends for resources on the Web server, rather than by modulating the Web workload itself.

The testbed consists of an Apache2 Web server running on a Linux 2.4 system with a 600 MHz CPU and 256 MB RAM. The Web server stores 2000 files of size 100 KB each. Each client session fetches randomly chosen files in sequence at the maximum rate; the random-access workload forces a fixed portion of the requests to access the disk. This leads to an average response time for the system of around 70 msec, with a normal throughput of about 90 file downloads per second.

We analyzed data from four test runs, each with a competing "resource hog" process on the Web server:

Disk The process writes a stream of data to the Web server disk for 10 minutes of a 20 minute run.

Mem The process contends for memory for 2.5 hours of a 10.5 hour run.

I/O A scheduled backup occurs during the run. The test runs for 14.5 hours; the backup takes about an hour.

CPU The process contends for CPU throughout the run.

We used a single SLO threshold derived from the system response time without contention. For each test the

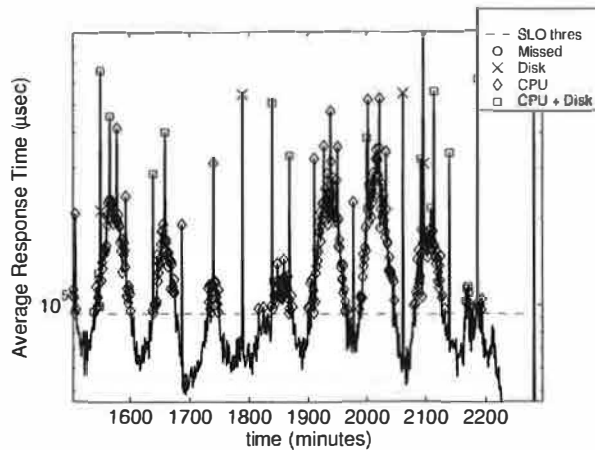


Figure 11: Plot of average response time for RAMP experiment with instances of violation marked as they are explained by different combinations of the model metrics. The horizontal line shows the SLO threshold.

exper.	SLO th (msec)	TAN BA	TAN Det	TAN FA
Disk	204	92.5 ± 12.9	88.3 ± 15.3	3.3 ± 10.5
Mem	98	99.5 ± 0.3	99.6 ± 0.01	0.6 ± 0.6
I/O	73	97.9 ± 1.4	97.8 ± 2.4	1.9 ± 0.4

Table 6: Summary of results from the OpenView testbed.

system learns a TAN model based on 54 system-level metrics collected using SAR at 15 second intervals. We omit detailed results for the CPU test: for this test the induced models obtained 100% accuracy using only CPU metrics. Table 6 summarizes the accuracy of the TAN models for the other three tests.

Table 7 shows the metrics selected for the TAN models for each test. We see that for the Memory and I/O bottleneck tests, the TAN algorithm selected metrics that point directly to the bottleneck. The metrics for the disk bottleneck experiment are more puzzling. One of the metrics is the one-minute average load (`loadavg1`), which counts the average number of jobs active over the previous minute, including jobs that have been queued waiting for I/O. Since disk metrics were not recorded in this test, and since the file operations did not cause any unusual CPU, network, memory or I/O load, this metric serves as a proxy for the disk queues. To pinpoint the cause of the performance problem in this case, it is necessary also to notice that CPU utilization dropped while load average increased. With both pieces of information one can conclude that the bottleneck is I/O-related.

Disk

`ldavg-1`: System load average for the last minute
`plist-sz`: Number of processes in the process list

Mem

`pgpgout/s`: Total number of blocks the system
 paged out to disk per sec
`txpck/s`: Total number of packets transmitted
 per sec (On the eth0 device)

I/O

`tps`: Transfers per second on I/O device
`activepg`: Number of active (recently touched)
 pages in memory
`kbbuffers`: Amount of memory used as buffers
 by the kernel in kilobytes
`kbswpfree`: Amount of free swap space in kilobytes
`totsck`: Total number of sockets used

Table 7: Metrics selected in each of the three experiments, with short descriptions (from SAR man page).

These results provide further evidence that the analysis and TAN models suggest the causes of performance problems, either directly or indirectly, depending on the metrics recorded.

7 Related Work

Jain's classic text on performance analysis [25] surveys a wide range of analytical approaches for performance modeling, bottleneck analysis, and performance diagnosis. Classical analytical models are based on *a priori* knowledge from human experts; statistical analysis helps to parameterize the models, characterize workloads from observations, or selectively sample a space of designs or experiments. In contrast, we develop methods to induce performance models automatically from passive measurements alone. The purpose of these models is to identify the observed behaviors that correlate most strongly with application-level performance states. The observations may include but are not limited to workload measures and device measures.

More recent books aimed at practitioners consider goals closer to ours but pursue them using different approaches. For example, Cockcroft & Pettit [12] cover a range of facilities for system performance measurement and techniques for performance diagnosis. They also describe Virtual Adrian, a performance diagnosis package that encodes human expert knowledge in a rule base. For instance, the "RAM rule" applies heuristics to the virtual memory system's scan rate and reports RAM shortage if page residence times are too low. Whereas Virtual Adrian examines only system metrics, our approach correlates system metrics with application-level

performance and uses the latter as a conclusive measure of whether performance is acceptable. If it is, then our approach would *not* report a problem even if, e.g., the virtual memory system suffered from a RAM shortage. Similarly, Virtual Adrian might report that the system is healthy even if performance is unacceptable. Moreover, we propose to induce the rules relating performance measures to performance states automatically, to augment or replace the hand-crafted rule base. Automatic approaches can adapt more rapidly and at lower expense to changes in the system or its environment.

Other recent research seeks to replace human expert knowledge with relatively knowledge-lean analysis of passive measurements. Several projects focus on the problem of diagnosing distributed systems based on passive observations of communication among “black box” components, e.g., processes or Java J2EE beans implementing different tiers of a multi-tier Web service. Examples include WebMon [20], Magpie [4], and Pinpoint [10]. Aguilera *et al.* [2] provides an excellent review of these and related research efforts. It also proposes several algorithms to infer causal paths of messages related to individual high-level requests or transactions, and to analyze the occurrences of those paths statistically for performance debugging. Our approach is similar to these systems in that it relates application-level performance to hosts or software components as well as physical resources. The key difference is that we consider metrics collected within hosts rather than communication patterns among components; in this respect our approach is complementary.

Others are beginning to apply model-induction techniques from machine learning to a variety of systems problems. Mesiner *et al.* [27], for instance, apply decision-tree classifiers to predict properties of files (e.g., access patterns) based on creation-time attributes (e.g., names and permissions). They report that accurate models can be induced for this classification problem, but that models from one production environment may not be well-suited to other environments; thus an adaptive approach is necessary.

8 Conclusion

TANs and other statistical learning techniques are attractive for self-managing systems because they build system models automatically with no *a priori* knowledge of system structure or workload characteristics. Thus these techniques—and the conclusions of this study—can generalize to a wide range of systems and conditions. This paper shows that TANs are powerful enough to capture the performance behavior of a representative three-tier Web service, and demonstrate their value in sifting through instrumentation data to “zero in” on the most rel-

evant metrics. It also shows that TANs are practical: they are efficient to represent and evaluate, and they are interpretable and modifiable. This combination of properties makes TANs particularly promising relative to other statistical learning approaches.

One focus of our continuing work is online adaptation of the models to respond to changing conditions. Research on adapting Bayesian networks to incoming data has yielded practical approaches [22, 6, 19]. For example, known statistical techniques for sequential update are sufficient to adapt the model parameters. However, adapting the model structure requires a search over a space of candidate models [19]. The constrained tree structure of TANs makes this search tractable, and TAN model induction is relatively cheap. These properties suggest that effective online adaptation to a continuous stream of instrumentation data may well be feasible. We are also working to “close the loop” for automated diagnosis and performance control. To this end, we are investigating forecasting techniques to predict the likely duration and severity of impending violations; a control policy needs this information to balance competing goals. We believe that ultimately the most successful approach for adaptive self-managing systems will combine *a priori* models (e.g., from queuing theory) with automatically induced models. Bayesian networks—and TANs in particular—are a promising technology to achieve this fusion of domain knowledge with statistical learning from data.

9 Acknowledgments

Joern Schimmelpfeng and Klaus Wurster conducted the experiments described in Section 6. Sharad Singhal provided useful comments on a previous version of the paper. Finally we would like to thank the anonymous reviewers and our shepherd Sam Madden; their comments and insights greatly improved the quality of this paper.

References

- [1] T. F. Abdelzaher, K. G. Shin, and N. Bhatti. Performance guarantees for Web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1):80–96, January 2002.
- [2] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, Oct. 2003.
- [3] G. A. Alvarez, E. Borowsky, S. Go, T. H. Oromer, R. BEcker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: An automated

- resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems (TOCS)*, 19(4):483–518, November 2001.
- [4] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: Online modelling and performance-aware systems. In *Proceedings of the Ninth Workshop on Hot Topics in Operating Systems (HotOS IX)*, May 2003.
- [5] Bayesian network classifier toolbox. <http://jbnc.sourceforge.net/>.
- [6] J. Binder, D. Koller, S. Russell, and K. Kanazawa. Adaptive probabilistic networks with hidden variables. *Machine Learning*, 29:213–244, 1997.
- [7] C. Bishop. *Neural Networks for Pattern Recognition*. Oxford, 1995.
- [8] R. Blake and J. Breese. Automating computer bottleneck detection with belief nets. In *Proc. 11th Conference on Uncertainty in Artificial Intelligence (UAI'95)*, Aug. 1995.
- [9] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic systems. In *Proc. 2002 Intl. Conf. on Dependable Systems and Networks*, pages 595–604, Washington, DC, June 2002.
- [10] M. Y. Chen, A. Accardi, E. Kiciman, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *Proceedings of the First Symposium on Networked System Design and Implementation (NSDI'04)*, Mar. 2004.
- [11] D. Clark, C. Partridge, J. C. Ramming, and J. Wroclawski. A knowledge plane for the Internet. In *Proceedings of ACM SIGCOMM*, August 2003.
- [12] A. Cockcroft and R. Pettit. *Sun Performance and Tuning*. Prentice Hall, 1998.
- [13] N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, March 2000.
- [14] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC)*, August 2001.
- [15] R. P. Doyle, O. Asad, W. Jin, J. S. Chase, and A. Vahdat. Model-based resource provisioning in a Web service utility. In *Proceedings of the Fourth USENIX Symposium on Internet Technologies and Systems (USITS)*, March 2003.
- [16] R. Duda and P. Hart. *Pattern Classification and Scene Analysis*. John Wiley and Sons, New York, 1973.
- [17] A. Fox and D. Patterson. Self-repairing computers. *Scientific American*, June 2003.
- [18] N. Friedman, D. Geiger, and M. Goldszmidt. Bayesian network classifiers. *Machine Learning*, 29:131–163, 1997.
- [19] N. Friedman and M. Goldszmidt. Sequential update of Bayesian network structure. In *Proc. 13th Conference on Uncertainty in Artificial Intelligence (UAI'97)*, August 1997.
- [20] P. K. Garg, M. Hao, C. Santos, H.-K. Tang, and A. Zhang. Web transaction analysis and optimization. Technical Report HPL-2002-45, Hewlett-Packard Labs, Mar. 2002.
- [21] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning*. Springer, 2001.
- [22] D. Heckerman, D. Geiger, and D. Chickering. Learning Bayesian networks: The combination of knowledge and statistical data. *Machine Learning*, 20:197–243, 1995.
- [23] R. Huebsch, J. M. Hellerstein, N. L. Boon, T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proceedings of 19th International Conference on Very Large Databases (VLDB)*, Sept. 2003.
- [24] R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.
- [25] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, 1991.
- [26] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.
- [27] M. Mesnier, E. Thereska, G. R. Ganger, D. Ellard, and M. Seltzer. File classification in self-* storage systems. In *Proceedings of the First International Conference on Autonomic Computing (ICAC-04)*, May 2004.
- [28] D. Mosberger and T. Jin. <http://perf>: A tool for measuring Web server performance. In *First Workshop on Internet Server Performance (WISP)*. HP Labs report HPL-98-61, June 1998.
- [29] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.
- [30] J. Quinlan. *C4.5 Programs for machine learning*. Morgan Kaufmann, 1993.
- [31] D. Sullivan. *Using probabilistic reasoning to automate software tuning*. PhD thesis, Harvard University, 2003.
- [32] R. van Renesse, K. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(2):164–206, May 2003.
- [33] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: An information plane for networked systems. In *Proceedings of ACM HotNets-II*, November 2003.
- [34] I. Witten and E. Frank. *Data Mining: Practical machine learning tools with Java implementations*. Morgan Kaufmann, 2000.

Automatic Misconfiguration Troubleshooting with *PeerPressure*

Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, Yi-Min Wang
{helenw, jplatt, ychen, ymwang}@microsoft.com
Microsoft Research

Abstract

Technical support contributes 17% of the total cost of ownership of today's desktop PCs [25]. An important element of technical support is troubleshooting misconfigured applications. Misconfiguration troubleshooting is particularly challenging, because configuration information is shared and altered by multiple applications.

In this paper, we present a novel troubleshooting system: *PeerPressure*, which uses statistics from a set of sample machines to diagnose the root-cause misconfigurations on a sick machine. This is in contrast with methods that require manual identification on a healthy machine for diagnosing misconfigurations [30]. The elimination of this manual operation makes a significant step towards automated misconfiguration troubleshooting.

In *PeerPressure*, we introduce a ranking metric for misconfiguration candidates. This metric is based on empirical Bayesian estimation. We have prototyped a *PeerPressure* troubleshooting system and used a database of 87 machine configuration snapshots to evaluate its performance. With 20 real-world troubleshooting cases, *PeerPressure* can effectively pinpoint the root-cause misconfigurations for 12 of these cases. For the remaining cases, *PeerPressure* significantly narrows down the number of root-cause candidates by three orders of magnitude.

1 Introduction

Today's desktop PCs have not only brought their users an enormous and ever-increasing number of features and services, but also an increasing amount of troubleshooting costs and productivity losses. Studies have shown that technical support contributes 17% of the total cost of ownership of today's desktop PCs [25]. A large amount of technical support time is spent on troubleshooting.

Many troubleshooting cases are due to misconfigurations. Such misconfiguration is often caused by data that is in shared persistent stores such as the Windows registry and UNIX resource files. These stores may serve many purposes. They include system-wide resources that are naturally shared by all applications (e.g., the file sys-

tem). They allow applications installed at different times to discover and integrate with one another. They enable users to customize default handlers or appearances of existing applications. They allow individual applications to register with system services to reuse base functionalities. They permit individual components to register with host applications that provide an extensibility mechanism (e.g., toolbars in browsers). To simplify our presentation, we will focus our discussion on a particular type of important configuration data: the Windows Registry [24], which provides hierarchical persistent storage for named, typed entries. Our discussions, techniques, and principles are directly applicable to other types of configuration stores, such as files, and other platforms, such as UNIX.

Misconfigurations can be introduced in many ways. For example, an application may unilaterally make seemingly innocuous changes to shared system configurations and cause unexpected behaviors in another application. A software bug may corrupt a Registry entry (by leaving a data field empty, for example) and break other programs that cannot handle an incorrect data format. Applying security patches may introduce Registry settings that are incompatible with existing applications [11]. Failed uninstallation of applications may introduce configuration inconsistencies resulting in malfunctions [17]. Administrators may inadvertently corrupt Registry entries when they try manually to fix misconfiguration problems using a Registry editor. Ganapathi et al. analyzed and categorized Registry misconfiguration problems based on their manifestation and scope of impact [14]. Some examples include missing Registry entries causing all network connections to disappear from the Control Panel, an extraneous entry causing a CD-ROM player to become inaccessible, a corrupted entry causing the system to log out a user upon any successful login.

Maintaining healthy configurations of a computer platform with a large installed base and numerous third-party software packages has been recognized as a daunting task [19]. The considerable number of possible configurations and the difficulty in specifying the "golden state" [26] (the perfect configuration) have made the problem appear to be intractable.

In this paper, we address the problem of misconfigu-

ration troubleshooting. There are two essential goals in designing such a troubleshooting system:

1. Troubleshooting effectiveness: the system should effectively identify a *small* set of sick configuration candidates in a short amount of time;
2. Automation: the system should minimize the number of manual steps and the number of users involved.

To diagnose misconfigurations of an application on a sick machine, it is natural to find a healthy machine to compare against [30]. Then, the configurations that differ between the healthy and the sick are misconfiguration suspects. However, it is difficult to identify a healthy machine *automatically*. Involving the user in confirming the correct application behavior seems unavoidable.

We can avoid extensive manual identification work by observing that *the golden state is in the mass*. In other words, an application functions correctly on *most* of machines, therefore we can use the statistics from a large enough sample set as the “statistical golden state”. The statistical golden state can be combined with Bayesian statistics to identify anomalous misconfigurations on sick machines. Then, the misconfigurations can be corrected by conforming to the majority of the samples. Hence, we name this statistical troubleshooting method *PeerPressure*.

We have prototyped a PeerPressure-based troubleshooting system which draws samples from a database of 87 real-usage machine configuration snapshots. We have evaluated the system with 20 real-world troubleshooting cases. PeerPressure can effectively pinpoint the root-cause misconfigurations for 12 of the cases. For the remaining ones, PeerPressure significantly narrows down the number of root-cause candidates by three orders of magnitude. These results have demonstrated PeerPressure as a promising troubleshooting method.

We will first give an overview of the architecture and operations of the PeerPressure troubleshooting system in Section 2. In Section 3, we detail the formulation and the analysis of the PeerPressure algorithm. We discuss our prototype implementation in Section 4. Then, we present our empirical results in Section 5. We compare and contrast our work with the related work in Section 6, address future work in Section 7, and finally conclude in Section 8.

2 The PeerPressure Architecture

Figure 1 illustrates the architecture and the operations of a PeerPressure troubleshooting system. A troubleshooting user first records the symptom of the faulty application execution on the sick machine with “App Tracer”. “App

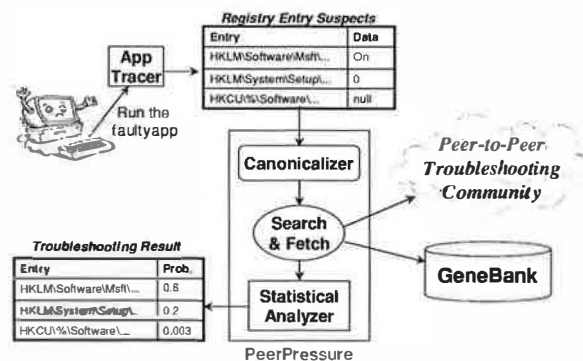


Figure 1: PeerPressure troubleshooting system architecture and its operations

Tracer” captures the registry entries that are used as input to the failed execution. These entries are the misconfiguration suspects. Then, the user feeds the suspects into the PeerPressure troubleshooter, which has three modules: a canonicalizer, a searcher/fetcher, and a statistical analyzer. The canonicalizer turns any user- or machine-specific entries into a *canonicalized* form. For example, user names and machine names are all replaced with constant strings “USERNAME” and “MACHINENAME”, respectively. Then, PeerPressure searches for a sample set of machines that run the same application. The search can be performed over a “GeneBank” database that consists of a large number of machine configuration snapshots or through a peer-to-peer troubleshooting community. In this paper, we base our discussions on the GeneBank database approach. For the peer-to-peer approach, we refer interested readers to [28]. Next, PeerPressure fetches the respective values of the canonicalized suspects from the sample set machines that also run the application under troubleshooting. The statistical analyzer then uses Bayesian estimation to calculate the probability for each suspect to be sick, and outputs a ranking report based on the sick probability. Finally, PeerPressure conducts trial-and-error fixing, by stepping down the ranking report and replacing the possibly sick value with the most popular value from the sample set. The fixing step interacts with the user to determine whether the sickness is cured: if not, configuration state needs to be properly rolled back. This last step is not shown in the figure and we will not address it for the rest of the paper.

As careful readers can see, there are still some manual steps involved. The first one is that the user must run the sick application to record the suspects. The second one is that the user is involved in determining whether the sickness is cured for the last step. We argue that these manual steps are difficult to eliminate because only the user can recognize the sickness, and therefore has to be in the loop for those steps. Nonetheless, these manual steps

only involve the troubleshooting user and not any second parties.

3 The PeerPressure Algorithm

In this section, we first illustrate the intuition and objectives for calculating the probability of a suspect being sick. Then, we derive the sick probability formula. At last, through our analysis, we show that our formulation achieves the objectives.

3.1 Intuition and Objectives

We use an example to illustrate the intuition and objectives of formulating the sick probability calculation for each suspect. Table 1 shows three suspects ($e1, e2, e3$) and their respective values from a sample set of machine configuration snapshots from the GeneBank. A cursory examination of the sample set suggests that $e1$ is probably healthy and $e2$ is more likely to be sick than $e3$. The suspect $e2$ is more likely to be sick because all samples have the same value, while the suspect value is different.

In fact, we have seen two types of state in canonicalized configuration entries: (I) application configuration states such as $e1$ and $e2$, (II) operational states such as timestamps, usage counts, caches, seeds for random number generators, window positions, and MRU (Most Recently Used)-related information. For troubleshooting configuration failures, we are mostly concerned with type I entries. Type II entries constitute the “natural biological diversity” among machines and are less likely to be root causes of configuration failures. In our example, $e3$ belongs to category II.

Therefore, the objective for the sick probability formulation is not only to capture the anomaly from the golden mass, but also to weed out the operational state false positives.

3.2 Formulation

Table 2 summarizes our notation.

To estimate whether a suspect is sick, we need to estimate $P(S|V)$, the probability that a suspect is sick given its value V . We estimate this probability for all suspects independently. In the derivation below, let us consider only one suspect i : all parameters are implicitly indexed by i .

According to Bayes rule [15], we have:

$$P(S|V) = \frac{P(V|S)P(S)}{P(V|S)P(S) + P(V|H)P(H)}. \quad (1)$$

We need to estimate each of the terms on the right-hand-side of Equation (1). We first assume that there is

only one sick entry amongst the suspects (leaving the multiple sick entry case for future work). Before we observe any values, the prior probabilities of a suspect being sick and healthy are

$$P(S) = \frac{1}{t}, \quad P(H) = 1 - \frac{1}{t},$$

where t is the number of possible suspects.

We do not have an extensive training set of sick suspects. Therefore, we assume that a sick entry has all possible values with equal probability:

$$P(V|S) = \frac{1}{c},$$

where c is the cardinality of the suspect entry, the total number of values that entry can take.

For $P(V|H)$, we leverage the observation of a sample set of machine configurations from the GeneBank. Let m denote the number of samples matching V , and N , the size of the sample set. If we assume that $P(V|H)$ is estimated via maximum likelihood, we get the estimate

$$P(V|H) = \frac{m}{N}, \quad (2)$$

$$P(S|V) = \frac{N}{N + cm(t-1)}. \quad (3)$$

However, maximum likelihood has undesirable properties when the amount of sample data is limited. For example, when there are no matching values to V in the sample set, then $m = 0$ and $P(S|V) = 1$, which expresses complete certainty that is unjustified. For example, in Table 1, maximum likelihood would claim that $e2$ and $e3$ are both sick with complete and equal confidence.

Bayesian estimation [15] of probabilities is more appropriate for the situation of small sample size N , such as our GeneBank scenario. Bayesian estimation uses a prior over $P(V|H)$, before the sample set is examined. The estimation then uses the posterior estimate of $P(V|H)$ after the sample set is examined. Therefore, $P(V|H)$ is never 0 or 1.

We first assume that $P(V|H)$ is multinomial over all possible values V . A multinomial is a probability distribution that governs a multi-sided die, where side j has label V_j attached to it. The probability of throwing the die and getting value V_j is p_j . Because one side is always up when a die is thrown, the p_j sum to one. The multi-sided die (and the multinomial) is completely characterized by the vector p_j .

We take the Bayesian approach of treating the p_j values as random variables. Therefore, the p_j themselves follow a distribution. Purely for mathematical simplicity, we assume that the p_j follow a Dirichlet distribution [15]. Dirichlet distributions are a natural prior for multinomials, because they are *conjugate* to multinomials. That is,

	Registry Key	Suspect Machine Registry Value	Machine 1 Value	Machine 2 Value	Machine 3 Value	Machine 4 Value	Machine 5 Value
e1	.jpg/contentType	image/jpeg	image/jpeg	image/jpeg	image/jpeg	image/jpeg	image/jpeg
e2	.htc/contentType	not exist	text/x-comp	text/x-comp	text/x-comp	text/x-comp	text/x-comp
e3	url-visited	yahoo	hotmail	nytimes	SFGate	google	friendster

Table 1: Intuition behind PeerPressure Sick Probability Formulation

combining observations from a multinomial with a prior Dirichlet yields a posterior Dirichlet. The use of Dirichlet priors is very common in Bayesian inference.

Dirichlet distributions are completely characterized by a count vector n_j , which corresponds to the number of possible counts for each value V_j . These counts do not need to reflect real observations: as we will see below, we can count phantom data, also.

To perform Bayesian estimation of $P(V|H)$, we first start by choosing a prior distribution. We assume that all values V_j are equally probable *a priori*. Therefore, we start by assuming a Dirichlet distribution with a count n for all possible values V_j . We then observe our N samples of values for this registry key, collecting counts m_j for the different values. The mean of the posterior Dirichlet yields the posterior estimate $P(V_j|H)$ [15]:

$$P(V_j|H) = \frac{m_j + n}{N + cn}. \quad (4)$$

The parameter n is proportional to the number of observations that are required to overwhelm the prior and to move the estimated $P(V|H)$ probabilities away from $1/c$. In other words, the higher the n is, the less confidence we have for the knowledge obtained from the GeneBank. The parameter n indicates the strength of the prior. A higher n leads to a stronger prior, which requires more evidence (observations N) to change the posterior.

We only need to estimate the $P(V_j|H)$ for the value that actually occurs in the suspect entry. Therefore, we can replace m_j with m , the number of samples that matches the suspect entry. Combining Equations (4) and (1) yields

$$P(S|V) = \frac{N + cn}{N + cnt + cm(t - 1)}. \quad (5)$$

Notice that Equation (5) never predicts a sick probability of zero or one, even if m is 0 or N , which is the whole point of using Bayesian statistics.

We choose $n = 1$ for our prior, which is equivalent to a flat prior: all multinomial values p_j are equally likely *a priori*. This is known as an “uninformative” prior.

Bayesian smoothing with an uninformative prior is just one method for smoothing probabilities. Another common method is the Turing-Good smoother [8], which also estimates the probability of a value unseen in the sample set. Unfortunately, Turing-Good is not easily applicable

N	Number of sample machines
t	Number of suspect registry keys
i	The index for the suspect key (from 1 to t)
V_i	The value of a suspect key i
c	Cardinality: the number of possible sample values for a suspect key
m	The number of samples that match the suspect value
$P(S)$	The prior probability that a suspect key is sick
$P(H)$	$1 - P(S)$
$P(S V)$	The probability that a suspect key is sick given its value
$P(V S)$	The probability that a sick suspect key i has value V_i

Table 2: Notation

for smoothing probability of registry values: Turing-Good requires a large number of distinct values which occur once or a few times in the sample set, in order to extrapolate the probability of a value that occurs zero times in the sample set. There are many registry values that have a small number of distinct values, hence the extrapolation performed by Turing-Good is not accurate. In other words, Turing-Good works well in linguistics, where the number of distinct values (words) is large, which is not always the case in the registry.

Because we do not use Turing-Good, we must handle the case of a value that is present in the suspect registry key, but is unseen in the sample values. We handle this by counting the number of distinct values in the sample set, c_0 , then adding an additional new value, called “unseen”. Any suspect value that is not present in the sample set will be assigned to the “unseen” class. Thus, we set the cardinality $c = c_0 + 1$ and compute the probability of a registry key being sick given a previously unseen value to be $cn/(cnt + cm(t - 1))$.

3.3 Asymptotic analysis

To show that our Bayesian probability estimates in Equation (5) produce sensible results, we illustrate the asymptotic behavior of the estimates in various cases.

Given a suspect set of size t , there are four variables that affect the sick probability ranking for the suspects: the number of matches m , the Dirichlet prior strength n , the sample set size N , and the cardinality c . Please note that N can vary among the suspects because of the canonicalized entries. For example, for a user-specific canonicalized entry, the number of samples is the number of users rather than the number of machines in the GeneBank; and a machine can have multiple users. Now, we analyze how each of these parameters affects the sick probability and whether the trend agrees with our objectives (see Section 3.1).

Fixing N , c , and n , as the number of matches m increases, the sick probability decreases, as desired:

$$\lim_{m \rightarrow \infty} P(S|V) = 0.$$

Fixing N , c , and m , as the prior strength n increases, we have

$$\lim_{n \rightarrow \infty} P(S|V) = \frac{1}{t} = P(S).$$

This means that conducting a statistical analysis over such a sample set is useless in this case. This makes sense, because when n reaches infinity, the prior has infinite strength, and therefore observations offer no additional knowledge.

For understanding the influence of N , we assume that as N grows, m also grows as fN , for some fraction f between zero and one. Therefore,

$$\lim_{N \rightarrow \infty} P(S|V) = \frac{1}{1 + cf(t-1)}.$$

Notice in the infinite data limit, the prior is completely “washed out”, and the higher c , f , or t is, the less likely an entry is to be sick. We also have, for $N = m = 0$,

$$\lim_{N \rightarrow 0} P(S|V) = \frac{1}{t} = P(S).$$

This is also accurate: when $N = 0$, we are unable to make any observations. In this case, the suspect set is the only factor that determines the sick probability.

To illustrate the impact of the cardinality c , we first note that $c \rightarrow \infty$ implies $N \rightarrow \infty$. So, applying the analysis for N above, we have

$$\lim_{c \rightarrow \infty, N \rightarrow \infty} P(S|V) = \lim_{c \rightarrow \infty} \frac{1}{1 + cf(t-1)} = 0.$$

This is desirable because when c is large, it represents a higher level of “biological diversity”, and therefore, being different is less likely due to some sickness.

Now, we examine the case of operational state where $m = 0$ most likely, we have

$$P(S|V) = \frac{N + cn}{N + tcn}.$$

Fixing N , the sick probability decreases with increased cardinality when there are no matches because the derivative of $P(S|V)$ with respect to c is negative when $t > 1$. When $t = 1$, $P(S|V) = 1$ as desired. Therefore, for our example in Table 1, Formula 5 will rank $e2$ sicker than $e3$, as desired.

In summary, our analysis demonstrates that Formula 5 achieves our objective of capturing anomalies and weeding out operational state false positives. Later, in Section 5, we further demonstrate through real-world troubleshooting cases that our PeerPressure algorithm is indeed effective.

4 The PeerPressure Prototype

We have prototyped the PeerPressure troubleshooting system as shown in Figure 1. We have created a GeneBank database using Microsoft SQL Server 2000 [10], which contains real-usage registry snapshots from 87 Windows XP desktop PCs. Approximately half of these snapshots are from three diverse organizations within Microsoft: Operations and Technology Group (OTG) Helpdesk in Colorado, MSR-Asia, and MSR-Redmond. The other half are from machines across Microsoft that were reported to have potential Registry problems.

We have implemented the PeerPressure troubleshooter in C# [22], which issues queries to the GeneBank to fetch the sample values and carries out the sick probability calculation (Section 3). We use a set of heuristics for canonicalizing user-specific and machine-specific configuration entries in the suspect set. One obstacle we encountered during our prototyping is that values for a specific registry entry across different machines are the same but with different representations. For example, 1, “#1”, and “1” all represent the same value. Nonetheless, the first one is an integer and the latter two are different string representations. Such inconsistent representations of the same data affect all parameter values needed by the sick probability calculation. We use heuristics to unify the different representations of the same data value. We call this procedure “data sanitization” for future reference. For example, one such heuristic is to find all entries that have more than one type. (Registry entries contain a “type” field). For a registry entry that has both numeric-typed and string-typed values among different registry snapshots, all string values are converted into numbers.

Our PeerPressure troubleshooter, although unoptimized in its present form, is already fast. We use a dual Intel Xeon 2.4GHz CPU workstation with 1 GB RAM to host the SQL server and to run the troubleshooter. On average, it takes less than 45 seconds to return a root-cause ranking report for suspect sets of thousands of entries. The response time generally grows with the number of sus-

Maximum registry size	333,193
Minimum registry size	77,517
Average registry size	198,376
Median registry size	198,608
Distinct canonicalized entries in GeneBank	1,476,665
Common canonicalized entries	43,913
Distinct entries data-sanitized	918,898

Table 3: Registry Characteristics

pects. Further analysis shows that database queries dominate the troubleshooting response time because we issue one query per suspect entry. Figure 2 shows the relationship between the response time and the number of suspects for the 20 troubleshooting cases under study. With aggressive database query batching, we anticipate that the response time can be greatly improved.

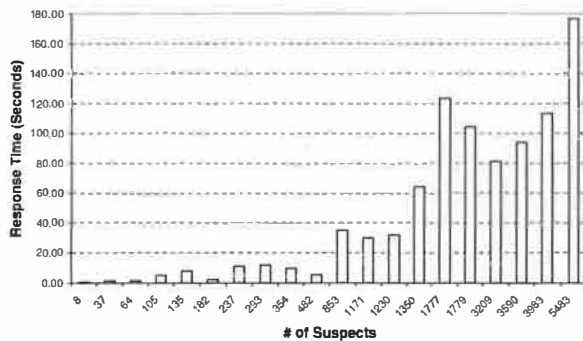


Figure 2: Response Time vs. Number of Suspects for 20 real-world troubleshooting cases.

5 Troubleshooting Effectiveness

In this section, we evaluate the troubleshooting effectiveness of the PeerPressure prototype on 20 real-world troubleshooting cases. We first examine the registry characteristics based on the registry snapshots from our GeneBank repository, then we present and analyze our troubleshooting results.

5.1 Registry Characteristics

The Windows Registry contains most of the configuration data for a desktop PC. Table 3 summarizes some registry statistics from the GeneBank. The sheer volume of configuration data is daunting. Figure 3 shows the registry size distribution among the registry snapshots in the GeneBank. Registry sizes range from 77,517 to 333,193 entries. The median is 198,608 entries. The total number of distinct canonicalized entries in the GeneBank is

1,476,665. Across all the machines, there are 43,913 common canonicalized entries. With our canonicalization heuristics, an average of 68,126 entries from each registry snapshot are canonicalized. With our data sanitization heuristics (see Section 4), we have sanitized 918,898 entries in the GeneBank.

Cardinality is an essential parameter of our PeerPressure algorithm (Section 3). Because the GeneBank may not contain all possible “genes” (entry values), we treat all values that are unknown to the GeneBank as a single value *unseen*. This unseen value effectively increments the observed cardinality from the GeneBank by one. Therefore, any entry from the GeneBank has a cardinality of at least two; and entries that do not exist in the GeneBank have a cardinality of one. In addition, some entries may not exist on some sample machines. For such cases, these entries have the value *no entry*. Figure 4 shows the distribution of the cardinality for all canonicalized entries in the GeneBank. 87% of the registry entries have a cardinality of two, 94% no more than three, and 97% no more than four. The low cardinality of registry entries contributes to the excellent troubleshooting results of our PeerPressure algorithm (as shown in the next section) because when the cardinality is low, the conformity among the samples is strong.

5.2 PeerPressure Performance with Real-World Troubleshooting Cases

In this section, we present our empirical troubleshooting results for PeerPressure.

We use the 20 cases listed in Table 4 for our experiments. They were all real-world failures that troubled some users. We have the knowledge of root-cause misconfiguration *a priori*. We picked 20 out of 129 accessible cases from Microsoft (MS) Product Support Service (PSS) e-mail logs, MS Helpdesk, web support forums, and our co-workers’ reported problems. The only criterion we used in selecting these cases is the ease of reproducing application failures since some cases require special hardware setup or specific software versions. Cases 1, 11, 13, and 14 are among the most commonly encountered configuration problems from MS PSS e-mail logs [14]; Cases 2, 9, and 10 are from MS Helpdesk; Cases 15-18 are from a web support forum; and the rest are from our co-workers.

Since we know the misconfigured, root-cause entry for each case, we use the ranking of the entry as our evaluation metric. To allow parameterized experiments, we reproduced these failures on a real-usage desktop using configuration user interface (e.g., Control Panel applets) to inject the failures whenever possible, and using direct editing of the Registry for the remaining cases. Then, we used “App Tracer” to get the suspects, the entries that are

ID	Name	Description of Problem
1	System Restore	No available checkpoints are displayed because the calendar control object cannot be started due to a missing Registry entry.
2	JPG	Right clicking on a JPG image and choosing the Send To → Mail Recipient option does not offer the resize option dialog box due to a missing Registry entry.
3	Outlook	User is always asked upon exiting Outlook whether she wants to permanently delete all emails in the Deleted Items folder, due to a hard-to-find setting.
4	IE Passwords	Internet Explorer (IE) browser does not offer to automatically save passwords; the option to re-enable the feature is difficult to find.
5	Media Player	Windows Media Player “Open Url” function fails if the EnableAutodial Registry entry is changed from 0 to 1 on a corporate desktop.
6	IM	MSN Instant Messenger (IM) significantly slows down if the firewall client is disabled on a corporate desktop.
7	IE Proxy	IE on a machine with a corporate proxy setting fails when the machine is connected to a home network.
8	IE Offline	IE “Work Offline” option may be automatically turned on without user knowledge; user is then be presented with a cached offline page instead of the default start page when launching IE.
9	Taskbar	IE windows are unexpectedly grouped under the Windows Explorer taskbar group, due to the addition of a Registry entry.
10	Network Connections	Control Panel → Network Connections shows nothing, due to a missing Registry key.
11	Folder Double-Clicking	Double clicking any folder in the right pane of Windows Explorer incorrectly brings up the “Search Results” window.
12	Outlook Express	Microsoft Outlook could not be started because the Outlook Express installation appears to be missing, due to a missing Registry key.
13	Cannot Start Executables	Double-clicking any EXE file does not launch the application.
14	Shortcut	Double-clicking any shortcut does not launch the application.
15	IE Menu Bar	IE menu bar disappears due to a corrupted Registry key name.
16	IE Favorites	IE uses the “unknown file type icon” for some of the links in the Favorites.
17	Sound Problem	Warning sound is missing when an invalid command was typed into Start → Run.
18	IE New Window	Right-clicking a link inside IE and choosing “Open in New Window” shows nothing.
19	Yahoo Toolbar	Yahoo Companion per-user installation affects all users.
20	Media Player in IE	Internet Explorer always launches Media Player on the left pane.

Table 4: 20 Real-World Troubleshooting Cases Used for PeerPressure Evaluation

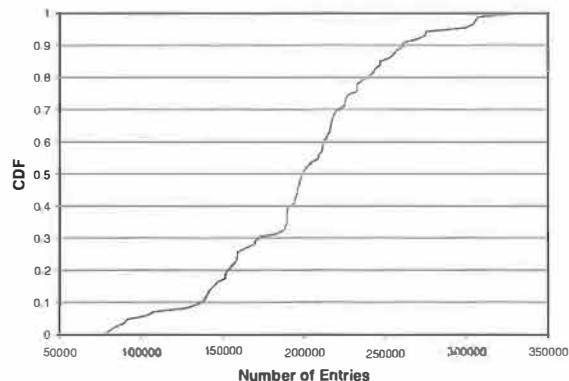


Figure 3: Registry Size Distribution

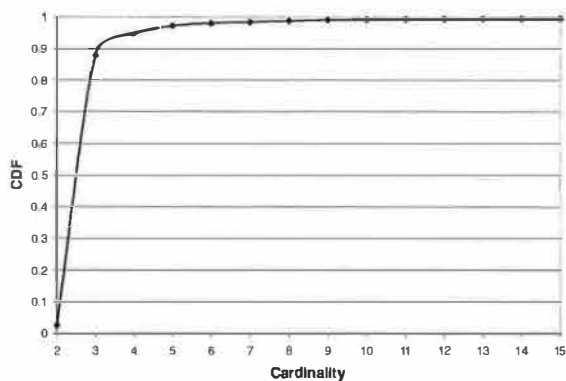


Figure 4: Cardinality Distribution

touched during the faulty execution of the application (see Section 2). Finally, we ran PeerPressure to produce the ranking reports.

5.2.1 Root Cause Ranking

For each troubleshooting case, Table 5 shows the ranking of the root-cause entry, the number of ties, the number of suspects, the cardinality of the root-cause entry, the number of samples matching the suspect's root-cause entry value, and the number of samples. Ranking ties can happen when the estimated probabilities of some entries have the same values. The non-zero values for the “# of Matches” column indicate that the GeneBank contains registry snapshots with the same sickness. Nonetheless, our assumption that the golden state is in the mass is still correct, since there are indeed only very small percentage of the sick machines in the GeneBank.

As we can see from the table, the number of suspects is large: ranging from 8 to 26,308, with a median of 1,171, and an average of 2,506. Therefore, PeerPressure is an indispensable step of troubleshooting since sieving through these large suspect sets for root-cause entries is like finding a needle in a haystack.

For 12 out of the 20 cases, PeerPressure ranks the root-cause entry as number one without any ties. For the remaining cases, PeerPressure narrows down the root-cause candidates in the suspect set by three orders of magnitude for most cases. There is only one case, case 19, which our GeneBank cannot help because only two machines in the GeneBank have the application and they happen to be sick and have the same sick values as well.

5.2.2 The Causes of False Positives

In this section, we give an analysis on the causes of false positives. The sick probability metric ranks the novelty of a suspect entry in the samples from the GeneBank. The more novel a suspect is compared with other suspects, the

lower its rank number is (i.e., the more sick the suspect is). One source of false positives is due to the nature of the root-cause entry. If the root-cause entry has a large cardinality, it likely receives a larger rank number based on our sick probability formula in Section 3. Case 20 falls into this category of false positives. The root-cause entry for Case 20 has a high cardinality of 65 while the rest of the cases have low cardinalities (Table 5).

The nature of the root-cause entry is only one factor. The ranking also depends on how the root-cause entry relates to other entries in the suspect set. A highly customized machine likely produces more noise, since the unique customizations can be even less conforming than a sick entry value. Case 11, 12, and 16 fall in this category.

Lastly, GeneBank is not pristine. The non-zero values in Column “# of Matches” in Table 4 indicate the number of machines in the GeneBank that have the same sickness. This affected the ranking of Case 2, 6, and 10.

5.2.3 The Impact of the Sample Set Size

It is intuitive that the larger the sample set is, and the better the root-cause ranking will be. However, our evaluation results indicate that this is not *entirely* true.

We have experimented with sample sets of size 5, 10, 20, 30, 50, and 87. For each sample set size N , we pick N samples from the GeneBank randomly for 5 times, then we average the root-cause ranking of the random sample sets. Table 6 shows root-cause ranking trend for various sample set sizes. The average number of ties for each sample set size is indicated in the parentheses. For the first three cases in the table, the root-cause ranking is perfect regardless of the sample set size. This perfect behavior is caused by all samples of the root-cause entry taking the same value. Any subset of the GeneBank samples still has the same value. No other suspects have a high sick probability when the sample set is small.

The cases belonging to the middle portion of Table 6

Case	Rank	Ties	# of Suspects	Cardinality	# of Matches	# of Samples
1. System Restore	1	0	1350	3	1	87
2. JPG	16	0	1779	3	5	87
3. Outlook	1	0	37	4	7	566
4. IE Passwords	1	0	135	4	1	566
5. Media Player	1	0	182	6	1	566
6. IM	12	0	1777	4	8	87
7. IE Proxy	1	0	1171	16	0	566
8. IE Offline	1	0	1230	4	1	566
9. Taskbar	1	0	64	4	2	566
10. Network Connections	2	0	354	2	1	87
11. Folder Double-Click	2	1	26308	2	0	87
12. Outlook Express	3	0	482	2	0	87
13. Cannot Start Executables	1	0	237	2	0	87
14. Shortcut	1	0	105	2	0	87
15. IE Menu bar	1	2	3590	2	0	87
16. IE Favorites	2	0	3209	3	0	87
17. Sound Problem	1	0	8	1	0	566
18. IE New Window	1	0	853	2	0	87
19. Yahoo Tool bar	n/a					
20. MediaPlayer in IE	9	0	5483	65	0	566

Table 5: Root-cause Ranking Results

Case	5 Samples (Ties)	10 (Ties)	20 (Ties)	30 (Ties)	50 (Ties)	87 (Ties)	# of matches
Perfect ranking regardless of the sample set size							
5. Media Player	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1
14. Invalid Shortcut	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	0
17. Sound Problem	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	0
Ranking Trend not solely dependent on the sample set size							
10. Network Connections	1.6 (1)	1.4 (0.6)	2 (0.2)	1.4 (0.2)	1.4 (0)	2 (0)	1
20. Media Player in IE	6.2 (0.2)	6.2 (0)	8 (0)	11 (0)	11.2 (0)	9 (0)	0
2. JPG	8.4 (0.2)	13.4 (0.4)	14.6 (0.2)	13 (0.2)	14.2 (0)	16 (0)	5
6. IM	15.6 (1.6)	104 (0.2)	20 (0)	15.4 (0)	14.6 (0)	8 (0)	8
Larger Sample Set improves ranking							
8. IE Offline	1 (0.2)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1
13. Cannot Start Executables	1 (0.4)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	0
1. System Restore	1 (0)	1 (0.2)	1 (0.2)	1 (0.2)	1 (0)	1 (0)	1
9. Taskbar	1.6 (5)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	2
3. Outlook	2.2 (0.4)	1.4 (0.8)	1.6 (0.8)	1.4 (0.8)	1 (0)	1 (0)	7
4. IE Passwords	5.8 (8.2)	3.2 (2.4)	3.2 (2.4)	1 (0)	1 (0)	1 (0)	1
7. IE Proxy	3.4 (1.8)	2.2 (0.2)	2 (0.8)	3(3.2)	1(0)	1 (0)	0
15. IE No Menu Bar	6.4 (10.8)	3.2 (3.6)	2.2 (2.6)	1.6 (2.4)	1 (2)	1 (2)	0
16. IE Favorites	18.2 (1)	3.8 (1.8)	3.2 (0.8)	3.8 (0)	2.8 (0)	2 (0)	0
18. IE New Window	7 (0.8)	3.8 (0.8)	2.2 (0)	1.6 (0)	1 (0)	1 (0)	0

Table 6: Impact of the Sample Set Size

Case	Machine 1	Machine 2	Machine 3
2	16 (0) / 1779	32 (2) / 1272	14 (0) / 1272
5	1 (0) / 182	1(0) / 566	1 (0) / 1657
6	1(0) / 2789	12(0) / 1777	12 (0) / 2017
14	1(0) / 105	1(0) / 84	1 (0) / 64
16	1 (0) / 302	2(0) / 3209	1 (3) / 1908

Table 7: Sick Machine Sensitivity Evaluation. Each entry has the format of RootCauseRanking (numberOfTies) / numberOfSuspects.

do not show a clear trend as a function of the sample set size. For Case 20, the root-cause entry has a scattered value distribution and a high cardinality of 65. Therefore, drawing any subset of the samples reflects the same value diversity, and therefore the ranking does not improve with larger sample set. For the other cases, although there is strong conformance in their value distributions, their rankings are affected by other entries in the suspect sets.

For the third category of the cases in the bottom part of Table 6, the root-cause ranking improves with larger sample set. For the first 4 cases, they have near-perfect root-cause ranking. Nonetheless, the number of ties decreases quickly as the sample set size increases. For most of the cases belonging to this category, we can see that the GeneBank has polluted entries according to the “# of Matches” column in Table 5. In this situation, enlarging the sample set reduces the impact of the polluted entries and therefore contributes to the decreasing trend of the rankings.

5.2.4 Sick Machine Sensitivity Evaluation

So far, we have only presented results from one sick machine’s vantage point. In fact, the troubleshooting results do depend on how uniquely the sick machine is customized and configured. To understand how our results vary with different sick machines, we have picked three real-usage machines that belong to different users, and evaluated the sick machine sensitivity with 5 cases. Table 7 shows that the troubleshooting results on these sick machines are mostly consistent. In some cases, such as Case 6, a larger suspect set leads to better ranking rather than introducing more noise, as one would have expected. This is simply because the larger suspect set on one machine is not necessarily a superset of the smaller suspect set on the other machine.

6 Related Work

There are two general approaches in system management: the white-box [6][3][9][21][16][27] and the black-box ap-

proach [30]. In the former, languages and tools are designed to allow developers or system administrators to specify “rules” of proper system behavior and configurations for monitoring and “actions” to correct any detected deviation. The biggest challenge for the white-box approach is in the accuracy and the completeness of the rule specification.

Strider [30], the precursor of this work, uses the black-box approach for misconfiguration troubleshooting: problems are diagnosed and corrected in the absence of specification of correct behavior. In Strider, the troubleshooting user first identifies a healthy machine on which the application functions correctly. This can be done by finding a healthy configuration snapshot in the past on the same machine or by finding a different healthy machine. Next, Strider performs configuration state differencing between the sick and the healthy, the difference is then further narrowed down by intersecting with suspects obtained from “App Tracer” (Section 2). Finally, Strider uses noise-filtering techniques to further narrow down the root-cause candidate set. Noise filtering uses a metric called *Inverse Change Frequency*, which looks at the change frequency of a registry entry. The more frequent an entry changes, the more likely it is a piece of operational state that is unlikely to be a root cause.

PeerPressure also takes the general black-box approach. PeerPressure differs from Strider in the following ways:

1. With statistical analysis, PeerPressure eliminates the manual step of the troubleshooting user identifying a healthy machine. This also eliminates the involvement of any second parties in cross-machine troubleshooting scenarios.
2. PeerPressure replaces the state-differencing and noise-filtering steps of Strider with a more general step of statistical analysis.
3. Strider uses order ranking which means that the final ordering of suspects is based on the sequence of their usage during application execution. The later the root-cause entry appears during the execution, the more false positives there are. In contrast, PeerPressure is not sensitive to the sequence of suspect entry usage. Nonetheless, the larger the suspect set is, the more likely there are entries that are more unique than the root-cause entry.
4. On the measure of root-cause ranking, PeerPressure’s yields better ranking for most of the cases.

Another interesting work that also takes the black-box approach is that of Aguilera et al. [1]. They address

the problem of black-box performance debugging for distributed systems. They developed and compared two algorithms for inferring the dominant causal paths. One uses the timing information from RPC messages. The other uses signal processing techniques. The significant finding of this work is that traces gathered with little or no knowledge of application design or message semantics are sufficient to make useful attributions of the sources of system latency. Therefore, their techniques are applicable to almost any distributed systems.

In a recent position paper, Redstone et al. [23] described a vision of an automated problem diagnosis system that automatically captures aspects of a computer's state, behavior, and symptoms necessary to characterize the problem, and matches such information against problem reports stored in a structured database. Redstone's work addresses the troubles with *known* root causes. PeerPressure complements this work with the techniques that identify the root causes of unsolved troubleshooting cases.

The concept of using statistical techniques for problem identification has emerged in several areas in recent years. One way of using statistics is to build a statistical model of healthy machines, and compare a sick machine against the statistical model. PeerPressure falls into this category and is the first to apply Bayesian techniques to the problem of misconfiguration troubleshooting. Other related work in this category [12][18][13] first use statistics to build a correct behavior model which is used to detect anomalies. Then, the number of false positives is minimized as much as possible. Engler et al. [12] use static analysis on the source code to derive likely invariants based on the statistics on some pre-defined rule templates (such as a call to *function a()* must be paired with a call to *function b()*). Then, potential bugs are recognized as deviant behaviors from these invariants. Engler et al. have discovered hundreds of bugs in Linux and FreeBSD to date. Later, they further improved the false positive rate in [18]. Forrest et al.'s seminal work on host-based intrusion detection system [13] builds a normal-behaving system call sequence database by observing system calls for various processes. Then, the intrusions with abnormal system call sequence can be caught. Apap et al. [2] designed a host-based intrusion detection system that builds a model of normal Registry behavior through training and showed that anomaly detection against the model can identify malicious activities with relatively high accuracy and low false positive rate.

Another way of using statistics is to correlate the observed service failure with root-cause software components or source code for debugging. Liblit et al. [20] uses statistical sampling combined with a number of elimination heuristics to analyze program behaviors. Program failures, such as crashes, are correlated with specific features or even specific variables in a program.

Brown et al [5] takes a black-box approach to infer hardware or software component dependencies by actively probing a system. Statistical model is then used to estimate dependency strengths. Similarly, Brodie et al [4] uses active probing with network events like *pings* or *traceroutes* for diagnosing network problems in a distributed system: a small set of high quality probes are first selected, then a Bayesian network is constructed for inferring faults.

The PinPoint root-cause analysis framework [7] is a debugger for component-based systems. PinPoint identifies individual faulty components that cause service failures in a distributed system. PinPoint uses data clustering on a large number of multi-tier request-response traces that are tagged with perceived success/failure status. The clustering determines the root-cause subset component(s) for the service failures.

7 Future Work and Discussion

We have interesting future work ahead of us. In this paper, we assume that there is only one sick entry among the suspects. However, it is possible that multiple entries contribute to the sickness collectively. We call the process of identifying multiple root-cause entries, *multi-gene troubleshooting*. Determining the number of genes involved in a troubleshooting case in addition to formulating the multi-gene sick probability are non-trivial tasks because the sick probability of entries are no longer independent.

Sometimes, a failed application execution may be caused by another application's misconfigurations. For example, a web browser may fail because of an incorrect VPN setting. The trace obtained from our AppTracer would not contain root-cause misconfigurations of another application. Cross-application misconfiguration troubleshooting remains an open challenge.

In environments where most or all machines have automatically maintained configurations, sample set selection criteria would need to be adjusted not to include the machines under the same automatic management.

PeerPressure takes the advantage of the strong conformance in most of the configuration entries for diagnosing the anomalies. However, some technology savvy users may customize their PCs so heavily that their configurations appear unique or "anomalous" to PeerPressure. These are inevitable false positives produced by PeerPressure.

Another open question is GeneBank maintenance. The GeneBank currently has one-time machine configuration snapshots from 87 volunteers. Without further maintenance, these configuration snapshots will be essentially out-of-date because of numerous software and OS upgrades. Effectively managing the evolving GeneBank is a

challenge. Further, we have not yet addressed the privacy issue. The privacy for both the users who contribute their configuration snapshots to the GeneBank and the users who troubleshoot their computers with the GeneBank need to be protected for real deployment. An alternative to the GeneBank approach is to “search and fetch” in a peer-to-peer troubleshooting community (see Section 2). Drawing the sample set in a peer-to-peer fashion is essentially treating all computer configuration snapshots from all the peer-to-peer participants as a distributed database that is always up-to-date and requires no maintenance. Nonetheless, the peer-to-peer approach does result in longer search and response time. Further, ensuring the integrity of the troubleshooting result is a challenge in the face of unreliable or malicious peers. We have a proposal for a privacy and integrity-preserving peer-to-peer troubleshooting system. For details, please see [29]

8 Conclusions

We have presented PeerPressure, a novel troubleshooting system that uses statistics from a set of sample machines as the golden state to diagnose the root cause misconfigurations on a sick machine. In PeerPressure, we introduce a ranking metric based on Bayesian estimation of the probability of a suspect candidate being sick, given the value of that suspect candidate.

We have developed a PeerPressure troubleshooter and used a database of 87 real-usage machine configuration snapshots to evaluate its performance. With 20 real-world troubleshooting cases, PeerPressure can effectively pinpoint the root-cause misconfigurations for 12 of the cases. For the remaining cases, PeerPressure significantly narrows down the number of root-cause candidates by three orders of magnitude.

In addition to achieving the goal of effective troubleshooting, PeerPressure also makes a significant step towards automation in misconfiguration troubleshooting by using a statistical golden state, rather than manually identifying a single healthy state.

Future work includes multi-gene troubleshooting where there are multiple root-cause entries instead of one, as well as privacy-preservation mechanisms for real deployment.

9 Acknowledgements

We inherited the “App Tracer” that was implemented by Chad Verbowski for the Strider toolkit [30]. Chad has also given us valuable feedback and discussions on the GeneBank database schema design as well as on canonicalization heuristics for Registry entries. Emre Kiciman

has also contributed to the canonicalization heuristics design. Chris Meek has given us valuable suggestions on Bayesian estimation algorithms. We received much advice from our database experts (Venki Ganti, Zhiyuan Chen, and Nico Bruno) for the GeneBank design and optimizations. This work also benefited from numerous discussions with Chun Yuan and Zheng Zhang. Zheng Zhang has given us great support and encouragement on this work. Mike Jones and Atul Adya also gave us helpful comments on the paper and encouraged us to submit it to OSDI. The anonymous OSDI reviewers and our Shepherd Matt Welsh provided us detailed and insightful critiques. We thank everyone for their help.

References

- [1] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of SOSP* (2003).
- [2] APAP, F., HONIG, A., HERSHKOP, S., ESKIN, E., AND STOLFO, S. J. Detecting Malicious Software by Monitoring Anomalous Windows Registry Accesses. In *Proceedings of LISA* (1999).
- [3] BAILEY, E. Maximum RPM, 1997.
- [4] BRODIE, M., RISH, I., AND MA, S. Intelligent probing: A Cost-Efficient Approach to Fault Diagnosis in Computer Networks. *IBM Systems Journal* 41, 3 (2002).
- [5] BROWN, A., KAR, G., AND KELLER, A. An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in a Distributed Environment. In *Seventh IFIP/IEEE International Symposium on Integrated Network Management* (may 2001).
- [6] BURGESS, M. A Site Configuration Engine. In *Computer Systems* (1995).
- [7] CHEN, M., KICIMAN, E., FRATKIN, E., FOX, A., AND BREWER, E. Pinpoint: Problem Determination in Large, Dynamic, Internet Services. In *Proceedings of International Conference on Dependable Systems and Networks (IPDS Track)* (2002).
- [8] CHURCH, K., AND GALE, W. A comparison of the enhanced Good-Turing and deleted estimation methods for estimating probabilities in English bigrams. *Computer Speech and Language* 5 (1991), 19–54.

- [9] COUCH, A., AND GILFIX, M. It's Elementary, Dear Watson: Applying Logic Programming to Convergent System Management Processes. In *Proceedings of LISA* (1999).
- [10] DELANEY, K. *Inside Microsoft SQL Server 2000*. Microsoft Press, 2001.
- [11] DUNAGAN, J., ROUSSEV, R., DANIELS, B., JOHNSON, A., VERBOWSKI, C., AND WANG, Y.-M. Towards a Self-Managing Software Patching Process Using Black-Box Persistent-State Manifests. In *Proceedings of Int. Conf. on Autonomic Computing (ICAC)* (May 2004).
- [12] ENGLER, D., CHEN, D. Y., HALLEM, S., CHOU, A., AND CHELF, B. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)* (October 2001).
- [13] FORREST, S., HOFMEYR, S. A., SOMAYAJI, A., AND LONGSTAFF, T. A. A Sense of Self for UNIX Processes. In *Proceedings of the IEEE Symposium on Research in Security and Privacy* (1996).
- [14] GANAPATHI, A., WANG, Y.-M., LAO, N., AND WEN, J.-R. Why PCs Are Fragile and What We Can Do About It: A Study of Windows Registry Problems. In *Proceedings of IEEE Dependable Systems and Network (DSN)* (June 2004).
- [15] GELMAN, A., CARLIN, J., STERN, H., AND RUBIN, D. *Bayesian Data Analysis*. Chapman, 1995.
- [16] KELLER, A., AND ENSEL, C. An Approach for Managing Service Dependencies with XML and the Resource Description Framework. In *Journal of Network and Systems Management* (June 2002).
- [17] KICIMAN, E., AND WANG, Y.-M. Discovering Correctness Constraints for Self-Management of System Configuration". In *Proceedings of Int. Conf. on Autonomic Computing (ICAC)* (May 2004).
- [18] KREMENEK, T., AND ENGLER, D. Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. In *Proceedings of 10th Annual International Static Analysis Symposium* (June 2003).
- [19] LARSSON, M., AND CRNKOVIC, I. Configuration Management for Component-based Systems. In *Proceedings of International Conference on Software Engineering (ICSE)* (May 2001).
- [20] LIBLIT, B., AIKEN, A., ZHENG, A. X., AND JORDAN, M. I. Bug Isolation via Remote Program Sampling. In *Proceedings of Programming Language Design and Implementation (PLDI)* (2003).
- [21] OSTERLUND, R. PIKT: Problem Informant/Killer Tool. In *Proceedings of LISA* (2000).
- [22] PETZOLD, C. *Programming Windows with C# (Core Reference)*. Microsoft Press, 2002.
- [23] REDSTONE, J. A., SWIFT, M. M., AND BERSHAD, B. N. Using Computers to Diagnose Computer Problems. In *Proceedings of HotOS* (2003).
- [24] SOLOMON, D. A., AND RUSSINOVICH, M. *Inside Microsoft Windows 2000*, 3rd ed. Microsoft Press, September 2000.
- [25] Web-to-Host: Reducing the Total Cost of Ownership, The Tolly Group. http://www.wrq.com/assets/products/tolly_tco.pdf, May 2000.
- [26] TRAUGOTT, S., AND HUDDLESTON, J. Bootstrapping an Infrastructure. In *Proceedings of LISA* (1998).
- [27] Tripwire. <http://www.tripwire.com/>.
- [28] WANG, H. J., HU, Y.-C., YUAN, C., ZHANG, Z., AND WANG, Y.-M. Friends Troubleshooting Network, Towards Privacy-Preserving Automatic Troubleshooting. Tech. Rep. MSR-TR-2003-81, Microsoft Research, Redmond, WA, Nov 2003.
- [29] WANG, H. J., HU, Y.-C., YUAN, C., ZHANG, Z., AND WANG, Y.-M. Friends Troubleshooting Network: Towards Privacy-Preserving, Automatic Troubleshooting. In *IPTPS* (San Diego, CA, Feb 2004).
- [30] WANG, Y.-M., VERBOWSKI, C., DUNAGAN, J., CHEN, Y., WANG, H. J., YUAN, C., AND ZHANG, Z. STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support. In *Proceedings of LISA* (2003).

Using Magpie for request extraction and workload modelling

Paul Barham, Austin Donnelly, Rebecca Isaacs and Richard Mortier
{pbar,austind,risaacs,mort}@microsoft.com
Microsoft Research, Cambridge, UK.

Abstract

Tools to understand complex system behaviour are essential for many performance analysis and debugging tasks, yet there are many open research problems in their development. Magpie is a toolchain for automatically extracting a system's workload under realistic operating conditions. Using low-overhead instrumentation, we monitor the system to record fine-grained events generated by kernel, middleware and application components. The Magpie request extraction tool uses an application-specific event schema to correlate these events, and hence precisely capture the control flow and resource consumption of each and every request. By removing scheduling artefacts, whilst preserving causal dependencies, we obtain canonical request descriptions from which we can construct concise workload models suitable for performance prediction and change detection. In this paper we describe and evaluate the capability of Magpie to accurately extract requests and construct representative models of system behaviour.

1 Introduction

Tools to understand complex system behaviour are essential for many performance analysis and debugging tasks, yet few exist and there are many open research problems in their development. Magpie provides the ability to capture the control path and resource demands of application requests as they are serviced across components and machines in a distributed system. Extracting this per-request behaviour is useful in two ways. Firstly it gives a detailed picture of how a request was serviced, throwing light on questions such as what modules were touched and where was the time spent? Did the request cause a disk access or was data served from the cache? How much network traffic did the request generate? Secondly, the per-request data can be analyzed to construct concise workload models suitable for capacity planning,

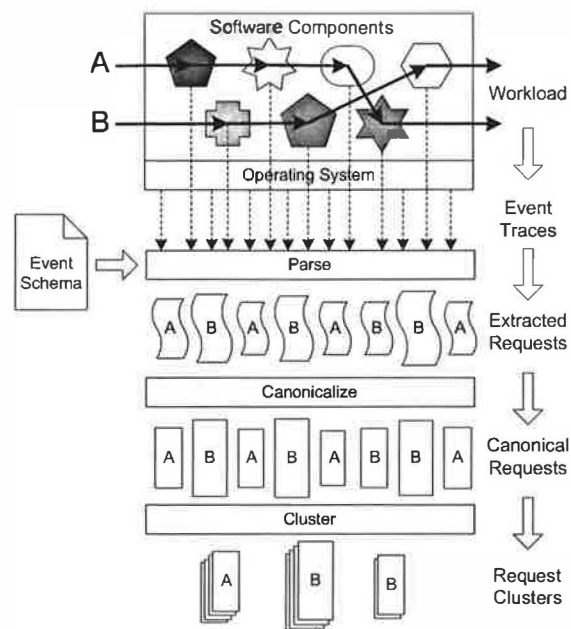


Figure 1: The Magpie toolchain: as requests move through the system event traces are generated on each machine. These are then processed to extract the control flow and resource usage by each individual request and scheduling variations removed. Finally, the canonical requests are clustered to construct models of the workload as a whole.

performance debugging and anomaly detection. These models require the ability to measure a request's resource demands, discarding the scheduling artefacts due to OS multitasking and timesharing. In effect, we obtain a picture of how the request *could* have been serviced (and apply this information toward modelling the workload), in addition to the data on how it *actually was* serviced (which is useful for detailed analysis of individual request behaviour).

The contributions of our work can be summarized as follows:

1. An unobtrusive and application-agnostic method

of extracting the resource consumption and control path of individual requests. Unlike other approaches to request tracking, for example [1, 8], Magpie does not require a unique request identifier to be propagated through the system, and it accurately attributes actual usage of CPU, disk and network to the appropriate request. This is achieved by correlating the events that were generated while the requests were live, using a schema to specify the event relationships and carrying out a temporal join over the event stream.

2. A mechanism for constructing a concise model of the workload. Each request is first expressed in a canonical form by abstracting away from the scheduling artefacts present in the original event trace. A representative set of request types is then identified by clustering the canonical request forms. This set of requests, together with their relative frequencies, is a compact model of the workload that can then be used for performance analysis purposes.
3. A validation of the accuracy of the extracted workload models using synthetic data, and an evaluation of their performance against realistic workloads.

The Magpie request tracking technique is unique in that it uses event logs collected in a realistic operating environment. It handles the interleaving of many different request types, it is impervious to unrelated activity taking place at the same time, and it is able to attribute resource usage to individual requests even when many are executing concurrently.

The request-oriented approach to understanding and characterizing system behaviour complements existing methods of performance modelling and analysis. Causes of faults or performance problems are often revealed simply by inspecting the Magpie trace of the individual request and comparing to the expected behaviour. In contrast, the traditional approach to monitoring system health is to log aggregate performance counters and raise alarms when certain thresholds are exceeded. This is effective for identifying some throughput problems, but will not catch others such as poor response time or incorrect behaviour (“why was the item not added to my shopping cart?”). Although straightforward programming errors and hardware failures are likely to be at the root of most problems, the effects are exacerbated and the causes obscured by the interactions of multiple machines and heterogeneous software components.

Even though performance modelling is of key importance for commercial enterprises such as data centers, current methods for constructing workload models are surprisingly unsophisticated. Without a tool like Magpie, workload models for capacity planning and other perfor-

mance prediction tasks have to be derived from a carefully controlled measurement environment in which the system input is contrived to stress each request type in isolation. This requires manual configuration and expert knowledge of the system behaviour, and compromises accuracy because variables such as caching behaviour are ignored. Workload models that are automatically derived using Magpie are quicker and easier to produce, and more accurately capture the resource demands of the constituent requests.

Magpie is a preliminary step towards systems that are robust, performance-aware and self-configuring (Autonomic Computing [12] is a well known articulation of this grand vision). We have previously discussed the applications and utility of Magpie’s workload models to scenarios ranging from capacity planning to on-line latency tuning [3, 11]. The emphasis in this paper is on a thorough, bottom-up evaluation of its use in practical situations. We demonstrate that Magpie accurately extracts individual requests under realistic operating conditions, and that the aggregation of this data leads to representative workload models.

The following four sections describe the design and implementation of the Magpie prototype toolchain. Then in Section 6 we evaluate the Magpie approach using simple synthetic workloads where it is straightforward to assess the results obtained, progressing to more complex workloads in Section 7.

2 Design and implementation

The workload of a system is comprised of various categories of request that will often take different paths through the system, exercising a different set of components and consuming differing amounts of system resources. A **request** is system-wide activity that takes place in response to any external stimulus of the application(s) being traced. For example, the stimulus of an HTTP request may trigger the opening of a file locally or the execution of multiple database queries on one or more remote machines, all of which should be accounted to the HTTP request. In other application scenarios, the database queries may be considered requests in their own right. Within Magpie both the functional progress of the request and its resource consumption at every stage are recorded. Thus a request is described by its path taken through the system components (which may involve parallel branches) together with its usage of CPU, disk accesses and network bandwidth.

The Magpie prototype consists of a set of tools that take event logs and eventually produce one or more workload models. Figure 1 illustrates the process. The intention when designing the tools has been to deploy an *online* version of Magpie that monitors request behaviour

in a live system, constantly updating a model of the current workload. Although Magpie operates both offline and online, this goal has dictated our design choices in many places.

Earlier versions of Magpie generated a unique identifier when a request arrived into the system and propagated it from one component to another [3]. The same technique is employed in other request tracking technologies such as Pinpoint [8]. Events were then logged by each component annotated with this identifier. We have since developed less invasive request extraction techniques that we describe in more detail below. Eschewing a requirement for global identifiers avoids the problems associated with guaranteeing unique identifier allocation. It also avoids the need for complicated ad-hoc state management or API modification to manage the identifiers as they are propagated. Finally, it also ensures that the instrumentation is kept independent of the definition of a “request”: it is not uncommon for two applications to share the same component, and it is desirable if one set of instrumentation can support tracing of both applications.

2.1 Instrumentation

The instrumentation framework must support accurate accounting of resource usage between instrumentation points to enable multiple requests sharing a single resource to be distinguished (e.g. threads sharing the CPU, RPCs sharing a socket). One consequence of this is a requirement for high precision timestamps. As events are generated by components in both user-space and kernel mode, the attribution of events to requests relies on them being properly ordered. In Windows NT based operating systems, a large proportion of kernel and device driver activity occurs inside Deferred Procedure Calls (DPCs); a form of software interrupt with a higher priority than all normal threads. It is therefore often important to know whether a particular event occurred inside a DPC or standard interrupt, or whether it occurred before or after a context switch. In order to get the required precision we use the processor cycle counter, which is strictly monotonic, as the event timestamp.

Event Tracing for Windows (ETW) [17, 18] is a low overhead event logging infrastructure built into recent versions of the Windows operating system, and is the technology underpinning the Magpie instrumentation. We make extensive use of pre-existing ETW event providers and where necessary we have added custom event tracing to components. The instrumented components in an e-commerce site that we use for prototyping are depicted in Figure 2. There are three main parts to the instrumentation:

1. Kernel ETW tracing supports accounting of thread

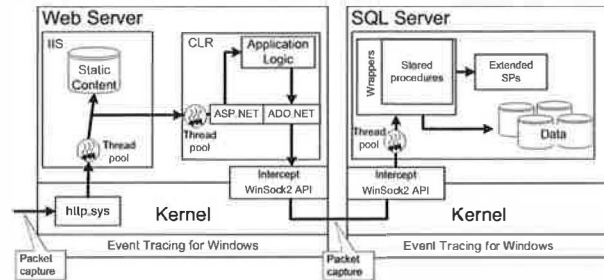


Figure 2: Instrumentation points for the web server and database server in our test e-commerce site. Some components such as the http.sys kernel module and the IIS process generate events for request arrival, parsing, etc. Additional instrumentation inserted by Magpie (shown in gray) also generates events; all these events are logged by the Event Tracing for Windows subsystem.

CPU consumption and disk I/O to requests.

2. The WinPcap packet capture library[19], modified to post ETW events, captures transmitted and received packets at each machine.
3. Application and middleware instrumentation covers all points where resources can be multiplexed or demultiplexed, and where the flow of control can transfer between components. In the prototype both platform middleware components such as WinSock2, and specific application-level components such as the ASP.NET ISAPI filter (used to generate active content), are instrumented in order to track a request from end to end.

An ETW **event** consists of a timestamp, an event identifier, and the values of zero or more typed *attributes*¹. In a typical system there will be multiple event providers, and therefore event identifiers have the hierarchical form *Provider/EventName*. A typical event from the log has the form:

Time, Provider/EventName, Attr1=Value1, ...

Each ETW event provider produces an ordered stream of timestamped events. However, at any given time there will usually be a large number of requests present in the system, each generating events from a variety of components and subsystems as it is serviced. As a result the stream of events will invariably comprise a non-deterministic interleaving of events from many active requests. The first stage of workload extraction is to demultiplex this event stream, accounting resource consumption to individual requests.

¹ All events with the same identifier have the same set of attributes.

2.2 Workload extraction pipeline

The toolchain consumes events generated by system instrumentation, as described in Section 2.1. In the sections following we present the workload extraction pipeline in some detail. The *request parser* identifies the events belonging to individual requests by applying a form of *temporal join* over the event stream, according to rules specified in an event schema. During this process it preserves the causal ordering of events, allowing a canonical form of each request to be inferred that captures its resource demands (as opposed to the service the request received), and this is discussed further in Section 4. From the canonical form, a request can be deterministically serialized, leading to a representation suitable for *behavioural clustering*. In Section 5 we describe how behavioural clustering builds workload models by comparing requests to each other according to both control path and resource demands.

3 Request parser

The request parser is responsible for extracting individual requests from the interleaved event logs. By determining which events pertain to a specific request, the parser builds up a description of the request that captures its flow of control and its resource usage at each stage. It is written to operate either online or offline, via the public ETW consumer API [17].

The parser considers each event from the stream in timestamp order and speculatively builds up sets of related events. It relies on an *event schema* to describe event relationships for the particular application of interest. For example, it may be the case that events occurring in sequence on the same thread belong to the same request, and this will be expressed in the schema by specifying those events related by thread id. The thread may post the identical event sequence for any number of different requests. The idea of *temporal joins* ensures that only those events that occur while the thread is working on the one request are joined together. Some of the resulting event sets will eventually be identified as describing a complete request, others can be discarded. Because the way in which events are related is defined out-of-band in an application-specific schema, the request parser itself contains no builtin assumptions about application or system behaviour.

3.1 Event schema

For every event type, the schema specifies which attributes connect it to other events. As each event is processed by the parser, its type is looked up in the schema and the event is then added to the appropriate set of re-

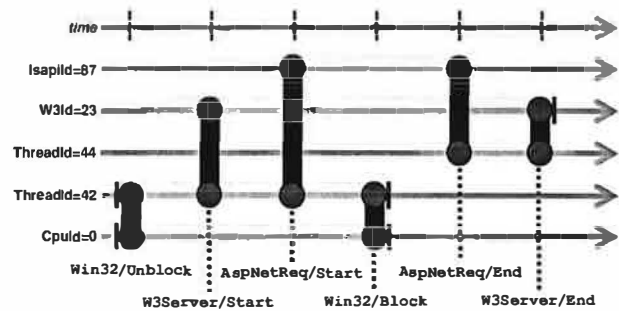


Figure 3: Illustration of how the parser joins a sequence of IIS events. Each event is shown as a black vertical line that binds two or more attribute-value pairs (represented as circles on the horizontal attribute-value lines). The joining of events is depicted with transparent gray lines, and valid-intervals are displayed with a vertical line to the left of the BIND_START and to the right of the BIND_STOP join attributes. This portion of the request does not show where the valid-interval for *W3Id=23* is opened, nor the opening or closing of the valid-interval for *ThreadId=44*.

```
EVENT("W3Server", "Start");
ATTRIBUTE("Tid", BIND_BASIC, 0);
ATTRIBUTE("W3Id", BIND_BASIC, 0);

EVENT("W3Server", "End");
ATTRIBUTE("Tid", BIND_BASIC, 0);
ATTRIBUTE("W3Id", BIND_STOP, 0);

EVENT("AspNetReq", "Start");
ATTRIBUTE("Tid", BIND_BASIC, 0);
ATTRIBUTE("IsapiId", BIND_BASIC, 0);

EVENT("AspNetReq", "End");
ATTRIBUTE("Tid", BIND_BASIC, 0);
ATTRIBUTE("IsapiId", BIND_BASIC, 0);

EVENT("Win32", "Unblock");
ATTRIBUTE("Tid", BIND_START, 0);
ATTRIBUTE("CpuId", BIND_START, 0);

EVENT("Win32", "Block");
ATTRIBUTE("CpuId", BIND_STOP, 0);
ATTRIBUTE("Tid", BIND_STOP, 0);
```

Figure 4: Portion of the IIS schema used to perform the event parsing illustrated in Figure 3. The binding types BIND.START and BIND.STOP instruct the parser to open or close a valid-interval.

lated events—in other words, the event is *joined* to one or more other events.

For example, an IIS web server schema specifies that one of the join attributes for both the *W3Server/Start* and the *W3Server/End* events is *W3Id*. This means that if two such events occur, both with *W3Id=23*, for example, they will be joined together. Figure 3 contains a graphical representation of this process. The same schema states that *ThreadId* is also a join attribute for those events. This allows different attributes posted by other event types to be transitively joined to the request for which *W3Id=23*. Thus, as shown in the diagram, if *AspNetReq/Start* with *IsapiId=87* is posted by the same *ThreadId=42* as *W3Server/Start*, then the two events will be joined

via the shared *ThreadId* attribute. In turn, the *IsapiId* join attribute causes other events also with *IsapiId*=87 to be added to this set of related events. In this way, the set of events belonging to each request is incrementally built up as the parser processes the event stream.

In addition to identifying which attributes cause events to be joined, the schema indicates the nature of these joins. In the example description above, there is nothing to stop two *W3Server/Start* events posted by the same thread but with different *W3Id* values being joined together. A mechanism is needed to prevent all the requests being merged into one, and this is captured by the notion of temporal joins.

3.2 Temporal joins

As a request progresses, relationships between attribute values are broken as well as created. For example, a worker from a thread pool may be re-tasked from one request to another, or an HTTP/1.1 connection may be reused for more than one request. In the above example *ThreadId*=42 is a worker thread that posts a *W3Server/Start* event on behalf of each request before handing the request processing off to another thread. The period during which we know that *ThreadId*=42 is working exclusively on one request defines a *valid-interval* for the attribute-value pair (*ThreadId*,42).

This terminology is borrowed from the temporal database community [10], where it is used to denote the time range during which a row of a table was present in the database. In such databases, arbitrary SQL queries can be executed against the database as if at a particular time. Theoretically it should be possible to implement the Magpie parser as queries against a temporal database in which each table holds the events of a given type. Finding all the events relating to a request would be an *n*-way relational join, where *n* is the number of event types involved in the request.

During a valid-interval, events are joined together as usual. However once the valid-interval is closed for a particular attribute-value pair, no more events can be added to the same event set via that pair. Therefore the IIS schema specifies that the event *Win32/Block* closes the valid-interval for the *ThreadId* attribute of that event, and likewise the *Win32/Unblock* event opens the valid-interval. In the example above, a new valid-interval for *ThreadId*=42 is opened for each request, thus preventing the merging of disjoint requests.

The opening and closing of valid-intervals is controlled in the schema by the use of *binding types* for the join specifications. A *BIND_START* begins a new valid-interval for an attribute-value pair and a *BIND_STOP* terminates the current valid-interval. An attribute that joins an event without affecting the valid-interval has

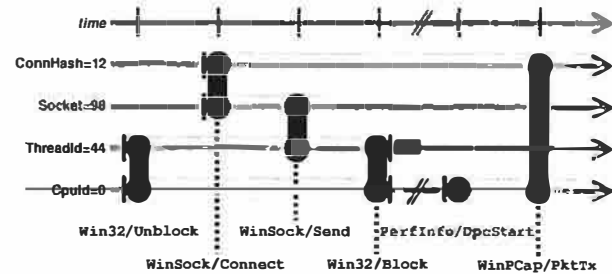


Figure 5: Transitive joins enable packet bandwidth and CPU consumption by the network stack to be correctly attributed, even though the thread that issues the send request is swapped out at the time the packet is transmitted. The diagonal pair of lines crossing the horizontal lines indicate the passing of an arbitrary amount of time.

a *BIND_BASIC* binding type. In the theoretical implementation using a temporal database, when a valid-interval is closed by an attribute-value pair, all the corresponding events would be deleted from the relevant table².

A fragment of the IIS schema matching the example discussed above is shown in Figure 4. In our prototype parser implementation, the schema is written as C macros. The *EVENT* macro takes the name of the provider and the event. Each join attribute is listed using the *ATTRIBUTE* macro, together with its binding type and any flags.

3.3 Resource accounting

Certain event types are associated with the consumption of physical resources. Specifically, context switch events give the number of CPU cycles used by a thread during its timeslice, disk read and write events are posted with the number of bytes read or written, and likewise packet events with the packet's size. When these event types are added to a request, it indicates that the relevant resource was consumed on behalf of that request. Figure 5 shows an example in which CPU consumption is associated with the request both while *ThreadId*=44 is running, and also during the network stack DPC (when some other thread is nominally swapped in on that CPU). The *WinPCap/PktTx* event also associates the network bandwidth used by that packet with the request. The user mode *WinSock/Connect* and kernel mode *WinPCap/PktTx* events are joined via their shared source and destination address attributes, represented in the diagram as *ConnHash*=12.

Figure 6 shows an annotated screenshot from a visualization tool that we developed to debug the parser. The highlighted sub-graph contains events from an HTTP request for the URL *shortspin.aspx*, which generates

²Hence the *n*-way relational join to find all events in a request would have to span multiple valid-intervals over multiple tables.

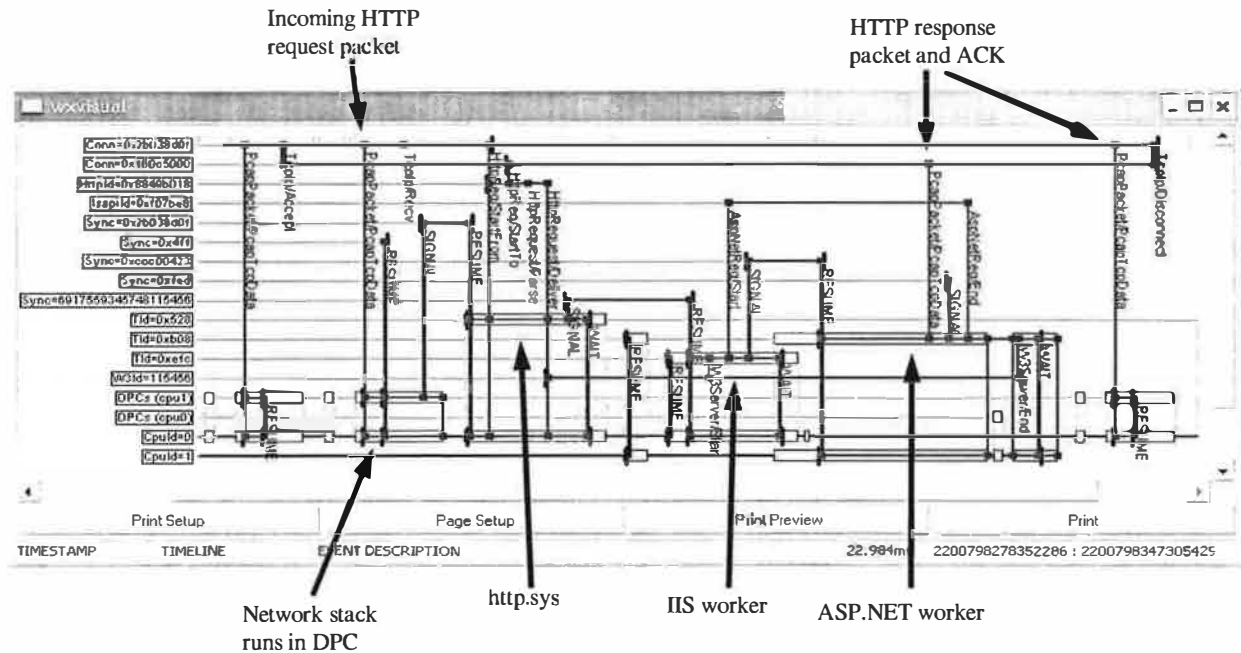


Figure 6: Annotated screenshot of parser visualization of a single request. Each of the event attribute-value pairs that is active during the displayed time period is depicted with a horizontal timeline. Events are shown as binding to one or more of these timelines, and when the binding is of type STOP or START, this is indicated with a small vertical barrier. The portions of each timeline that belong to the request are emphasized, showing that the parser is in effect doing a flood-fill of the graph formed by the join attributes of events. To make it easy to see which threads are actually running at any time, this is highlighted with a pale rectangle.

a very small amount of dynamic content that is returned in a single HTTP response packet. It also spins the CPU for approximately 15ms by executing a tight loop. This particular request is an example of a type B request as used in the experimental evaluation presented in Section 6.

3.4 Implementation

The design of the parser was severely constrained by the necessity for minimal CPU overhead and memory footprint, as it is intended to run *online* on production servers. Additionally, it must process trace events in the order they are delivered, since there is no way for it to seek ahead in the trace log, and this creates still more complexity.

In online mode, the kernel logger batches events from different subsystems for efficiency and delivers one buffer at a time. For example, context switch events from each CPU are delivered in separate buffers. Whilst individual buffers contain events in timestamp order, events from different buffers must be interleaved before processing. These reorderings are performed using a priority queue, and add approximately 1 second of pipeline delay to the parser.

The reorder queue is also used for some events that are posted at the end of the operation, such as those for disk I/O, which contain the start time of the I/O as an event

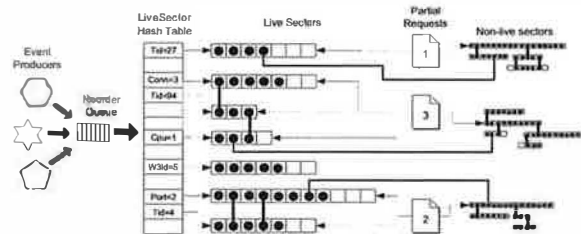


Figure 7: Parser data structures. Hash table entries represent the current valid-intervals and are known as *live sectors*. An event (shown as a black circle) is added to zero or more live sector lists according to the values of its binding attributes. Non-live sector lists, representing closed valid-intervals, are not reachable from the live sector hash table.

parameter. The parser creates a synthetic “Start” event with the correct timestamp and inserts it into the event stream in the correct place.

Figure 7 depicts the data structures used by the parser. Temporal joins are implemented by appending events to one or more time-ordered lists, each of which represents the current valid-interval for an attribute. The most recent (“live”) valid-interval for each attribute is maintained in a hash table and requests are gradually built up using event attribute bindings to connect lists together. In the example from the previous section, the parser would enter the *W3Server/Start* event onto the current list containing all events with *W3Id*=23 and onto the current list containing all events with *ThreadId*=42. The

presence of the same event in both lists causes them to be joined together, forming a larger sub-graph.

The schema identifies a seed event that occurs only and always within a request, for example a web server HTTP request start event. As events are processed in timestamp order, every so often some of the sub-graphs will become unreachable because all their valid-intervals have been closed. If a sub-graph contains a request seed event, then all the connected events can be output as a complete request, otherwise it will be garbage collected. Note that there will be many lists (or graphs) that turn out not to represent any request, but instead contain events generated by other applications and background system activities. An additional timeout is used to bound the resources consumed by such activities.

Ideally a request schema should be written by someone with an in-depth knowledge of the synchronization mechanisms and resource usage idioms of the application in question, and this would preferably be done at the same time as the application is instrumented. It is much harder to retrofit a schema and instrumentation to an existing application without this knowledge (but not impossible, as we have in fact done this for all the applications mentioned in this paper). An area for future work is to explore extensions to the expressiveness of the schema. Currently an event can only affect the timelines of its own attributes: one useful enhancement would be the ability to transitively start or stop the valid-intervals on other timelines.

3.4.1 Performance evaluation

We assessed the performance impact of Magpie event tracing and parsing by running a web stress-test benchmark. The number of HTTP requests served over a two-minute interval was recorded for a CPU-bound workload that generated active HTML content and saturated the CPU. Measurements were repeatable to within +/- 10 requests.

With no instrumentation the server was able to complete 16720 requests, i.e. 139 requests/second. When logging was turned on in real-time mode, with no event consumer, there was no discernible difference in throughput. A dummy event consumer, which immediately discarded every event, reduced the throughput to 136 requests/second. Running the Magpie parser to extract requests online resulted in a throughput of 134 requests/second, giving a performance overhead of approximately 4%, around half of which can be attributed to the ETW infrastructure. During these experiments the average CPU consumption of the parser was 3.5%, the peak memory footprint 8MB and some 1.2 million events were parsed. Since the web server was CPU-bound during the course of the experiments, this directly accounts for the

```
SYNC("TcpIp/Recv", "ConnHash", "HttpRequest/Start");
SYNC("HttpRequest/Deliver", "W3Id", "Win32/Unblock");
SYNC("AspNetReq/Start", "IsapiId", "Thread/Unblock");
SYNC("Thread/Exit", "Tid", "Thread/Join");
WAIT("Win32/Block");
```

Figure 8: Example statements from the IIS schema used to add explicit thread synchronization points to parsed HTTP requests. Each SYNC statement specifies (s, a, d) where s and d are the source and destination events, and are (transitively) joined by shared attribute a . An event pattern matching a SYNC specification will result in a Signal event being inserted on the source thread and a Resume event on the destination thread. A WAIT event type generates an additional synthetic Wait event.

observed drop in HTTP throughput.

When the ETW logging was enabled to write events to file (instead of operating in real-time mode), the server throughput was 138 requests/second, indicating that the impact of the ETW infrastructure in offline mode is negligible. For the same workload, of 2 minutes duration, a total of around 100MB of binary log file was produced. The parser extracted the correct number of requests in 5.6s, with a peak working set size of approximately 10MB. The average number of events attributed to each request was 36.

3.5 Synchronization and causal ordering

At the lowest level, all events are totally ordered by timestamp, leading to a trace of the execution and resource consumption that may vary depending on how the threads acting in a request happen to be scheduled. To extract a meaningful workload model we need to recover the threading structure within a request: for example, determining when one thread causes itself to block or another to unblock. This inter-thread causality tells us how much leeway the scheduler has to re-order the processing of the various stages of a request, and it also allows us to infer how portions of a request might parallelize, which is clearly of interest in multi-processor deployments.

In the web server example used for Figure 6, a kernel TcpIp/Recv event unblocks an IIS thread that parses HTTP requests, then unblocks an ISAPI filter thread, which eventually unblocks a third thread to run the ASP.NET active content generator. This last thread blocks after sending the HTTP response back to the client. Since these threads typically come from a thread pool, we occasionally observe the same thread processing multiple of these logically distinct segments of a request, so it is important to be aware of these synchronization points even if they are not apparent in all requests.

Many such synchronization points are implicit from the OS primitives being invoked (e.g. send and receive). In other places, thread synchronization can be performed using mechanisms for which there is no instrumentation,

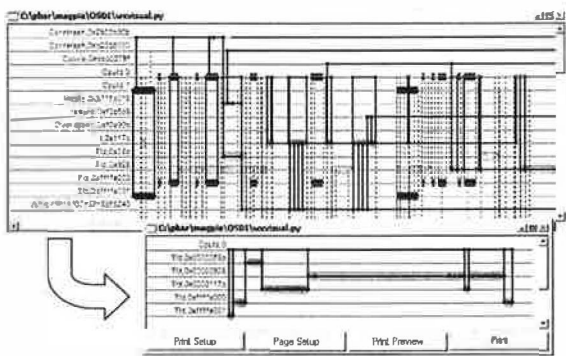


Figure 9: A canonical version of an HTTP request is produced by eliding all scheduling behaviour and retaining only thread synchronization points. The top window shows the request as scheduled in an experiment with 5 concurrent clients, and the lower window the canonical version.

e.g. in a user-level scheduler. For this reason, we provide a mechanism to explicitly insert causal dependencies into the parsed event graphs. This allows us to annotate a request with additional thread synchronization points using known semantics of the application domain.

We define three synthetic events to be inserted into parsed requests: Signal, Wait and Resume. There is an explicit causality between related Signal and Resume events, and so these will be connected by some shared attribute-value. This is expressed in the schema using a 3-tuple (s, a, d) , where s is the name of the source event executed by thread A at time t_A , d is the name of the destination event in thread B at time t_B , and a is the join attribute, shared by events s and d . Attribute a is not necessarily a parameter of both event types, but may be transitively shared through other joined events on the same thread. Events from thread A with timestamps less than t_A must always happen before thread B events with timestamps greater than t_B , under any scheduling discipline. Figure 8 shows some example synchronization statements from the IIS schema, and in Figure 6 the synchronization events inserted by the parser can be seen in amongst the original request events.

4 Canonicalization

When the system is more heavily loaded, requests tend to be scheduled in a highly interleaved fashion, as shown in Figure 9. Although the request URL is identical to that of Figure 6, the way in which the request is serviced differs due to multiple clients competing for system resources. In Figure 6, the thread completes its work in a single quantum of CPU time, whereas in the top window of Figure 9 it is frequently preempted and its activity is interspersed with threads servicing other connections.

A detailed per-request view of system activity is un-

doubtedly useful for determining the path taken by a request, and how it consumed resources along the way. However, for constructing workload models for performance prediction or debugging purposes we would rather represent requests as a *canonical* sequence of absolute resource demands and ignore all the information about how the request was actually serviced.

Using the causal ordering annotations discussed in the previous section, we produce a canonical version of each request, in effect by concatenating all resource demands between synchronization points, and then scheduling this as though on a machine with an unlimited number of CPUs. The lower window of Figure 9 shows the result of this processing stage when applied to the request in the upper window. The canonical version is clearly more useful for modelling purposes.

4.1 Cross-machine activity

When requests cause activity on multiple machines it is necessary to “stitch” together the events collected on each computer. Since every machine in the system has a separate clock domain for its timestamps, the request parser is run once for each machine, either online or offline (we believe it ought to be straightforward to extend the parser to deal with multiple clock domains but this is not a current priority). The request fragments from each machine are canonicalized as described previously. We then run an offline tool that combines canonical request fragments by connecting synchronization events from transmitted packets to received packets.

4.2 Request comparison

Thread synchronization can be used to overlay a binary tree structure onto what would otherwise be a linear timestamp-ordered event stream. When two threads synchronize, perhaps by sending a message, we create a logical fork in the event tree where the original (source) thread continues whilst also enabling the destination thread to execute in parallel. When a thread blocks, perhaps to receive a message, this is treated as a leaf node. The results of applying this procedure to a contrived RPC-style interaction is shown in Figure 10.

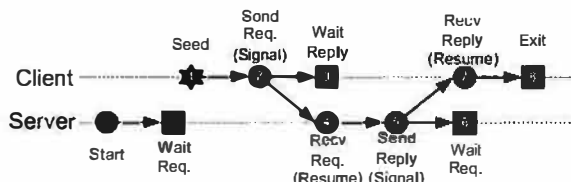


Figure 10: Binary tree structure overlaid onto RPC-style thread interactions. This tree would be deterministically serialized in the order shown.

By deterministically flattening this tree representation using standard depth-first traversal, we can cluster requests using a simple string-edit-distance metric rather than requiring elaborate and traditionally expensive graph-edit-distance metrics. Although this has produced reasonable results in our prototype, losing the tree structure before comparing requests seems likely to limit the usefulness of this approach in larger distributed systems where requests have more complex structure. Recent work has developed more suitably efficient algorithms for tree- and graph-edit-distance and also investigated graph-clustering [5]. Applying some of these techniques to improve our workload extraction process is currently under investigation.

5 Behavioural clustering

The clustering stage of the toolchain groups together requests with similar behaviour, from the perspective of both event ordering and resource consumption. Since we require that the processing pipeline functions online, we use a simple incremental clustering algorithm. The resulting clusters are the basis of a workload model which expresses that requests occur as typified by each cluster's representative, and they occur in proportion to their cluster's size.

The clusterer maintains a set of active workload clusters. For each cluster, we record a *representative* request (sometimes referred to as the centroid of the cluster), a cluster diameter, and the set of requests that are considered members of the cluster. Additionally, the algorithm keeps track of the average cluster diameter, and the average inter-cluster distance.

When a new request is presented to the clusterer, it computes the string-edit-distance between its serialized representation and that of each cluster centroid. The distance metric is a function of both the character edit cost (as in the traditional form of string-edit-distance) and also of the resource usage deltas associated with the two events. So for example, the comparison of two characters where both represent a disk read will give an edit cost proportional to the difference in how many bytes were actually read. The request is normally added to the cluster with the smallest edit distance, unless that edit distance exceeds a trigger threshold, in which case a new cluster is created.

6 Validation

To support our claim that Magpie is able to extract individual requests and construct representative workload models, we attempt to examine our techniques in isolation. In this section we present the results, which include

Type	URL	Resource
A	longspin.aspx	1 thread spins CPU 30ms
B	shortspin.aspx	1 thread spins CPU 15ms
C	small.gif	Retrieve static image 12Kb
D	tiny.gif	Retrieve static image 5Kb
E	parspin.aspx	2 threads spin CPU 15ms concurrently
F	rpcspin.aspx	1st thread spins CPU 7.5ms, signal other thread and wait, spin CPU 7.25ms 2nd thread spins CPU 15ms

Table 1: Request types and consumptions of primary resources.

an assessment of the quality of the clusters obtained, as well as checks that our resource accounting is accurate.

In all experiments, events are generated using the instrumentation described in Section 2.1 and the event streams are parsed as discussed in Section 3. Flattened representations of the requests are then clustered into similar groups using the behavioural clustering technique presented in Section 5. The machines used are all dual-processor 2.6GHz Intel P4, running Windows Server 2003, and IIS or SQL Server 2000. In all experiments we used Application Center Test [16] for the stress client.

We first evaluate the accuracy of Magpie's workload models using traces taken with a synthetic workload. In contrast to the more realistic operating conditions of Sections 7.1 and 7.2, the web site behaviour is calibrated to check that we extract the same workload as was injected. These experiments are intended to investigate the effectiveness of the request canonicalization mechanism, together with the behavioural clustering algorithm, to separate requests with different resource consumption.

The experiments were performed against a minimal ASP.NET website written in C#. Each URL consumes resources as depicted in Table 1. CPU is consumed within a tight loop of a fixed number of iterations, and network bandwidth is used by retrieving fixed size images.

6.1 Resource consumption

The string-edit-distance metric used by the clustering algorithm treats events annotated with their resource usage as points in a multi-dimensional space. To allow a more intuitive explanation of the resulting clusters, we first present results where resource consumption changes in only a single dimension. Concurrency is a complicating factor, and so we examine how well the request extraction and modelling tools perform, both under concurrent request invocations and when concurrent operations take place within a single request.

6.1.1 Processor time

We used the type A and type B requests of Table 1 (which differ only in their consumption of CPU cycles) to produce both single-varietal workloads and mixtures, using

CPU Resource Consumption

	Exp No	Workload / #Clients	Accntd CPU %	Clusters Found	Dia.	Min. Sep.	Model Error
Single	1.1	500A x1	96.6%	499 1 †	0.6 0.0	130 130	2.4%
	1.2	500B x1	94.0%	500	0.4	0.0	2.1%
	1.3	500A x5	96.6%	498 1 1 †	1.2 0.0 0.0	21.7 21.7 13.1	3.2%
Conc.	1.4	500B x5	94.6%	500	1.2	0.0	2.8%
	1.5	500(A/B) x1	95.9%	266A 234B	1.1 1.7	8.5 8.5	0.9%
	1.6	500(A/B) x5	95.9%	244A 254B 1A 1B †	1.2 1.2 0.0 0.0	8.0 8.0 33.2 103	2.0%

Table 2: Clusters produced from single and mixed request type workloads consuming CPU only, for both concurrent and serialized request invocations. *Accntd CPU %* is the fraction of CPU consumed by the relevant process that was accounted to individual requests. The *Clusters Found* column gives the number of requests found in each cluster (from a total of 500 requests). *Dia.* and *Min. Sep.* are the average cluster diameter and the distance to the centroid of the nearest other cluster, respectively. *Model Error* refers to the difference in resource consumption between the derived workload model and the parsed requests.

either serialized requests from a single client, or concurrent requests from 5 clients. This gives a total of 6 different workloads, as listed in the left-hand columns of Table 2. Note that even these trivial requests exhibit fairly complex interactions between the kernel and multiple worker threads—this is apparent in Figure 6, which depicts a type B request.

Table 2 records the clusters found from each of the workloads. It also shows the average distance from the cluster centroid of each request in the cluster (*Dia.*) and the distance to the centroid of the nearest other cluster (*Min. Sep.*). In experiments 1.1 and 1.2, 500 requests of type A (`longspin.aspx`) and 500 requests of type B (`shortspin.aspx`) each produced a large cluster. Repeating these experiments with 5 concurrent stress clients produced very similar results. The clusters produced under a concurrent workload generally have a larger internal diameter than those for the serial workload. Examining the distributions of total CPU consumed by the individual requests shows that, as predicted by the clusters, the concurrent workloads exhibit slightly larger spreads, probably due to cache effects and scheduling overheads.

As a validation check, the total amount of CPU time accounted to requests by the parser was summed and compared against the aggregate CPU times consumed by processes and the kernel during the course of each experiment. In Table 2, the column entitled *Accntd CPU %* indicates the fraction of “relevant” CPU that was accounted as request activity. Relevant CPU excludes that used by background processes and the idle thread, but includes most kernel mode activity and all user mode processing by the web server threads (this information is directly

Network Resource Consumption

	Exp No	Workload / #Clients	Accntd CPU %	Clusters Found	Dia.	Min. Sep.	Model Error
Single	2.1	500C x1	91.4%	500	0.1	0.0	12%
	2.2	500D x1	73.6%	500	0.1	0.0	8.3%
	2.3	500C x5	86.6%	498 2 †	0.9 2.1	15.3 15.3	20%
Conc.	2.4	500D x5	76.1%	498 1 †	0.9 0.0	9.8 9.8	14%
	2.5	500(C/D) x1	81.5%	246C 254D	0.1 0.1	6.9 6.9	19%
	2.6	500(C/D) x5	83.6%	267C 223D 10C	1.0 0.7 1.7	4.4 6.9 4.4	18%

Table 3: Clusters produced from single and mixed request type workloads differing primarily in consumption of network bandwidth, for both concurrent and serialized request invocations. See the Table 2 caption for explanation of the column headings.

available from the ETW event logs.) The reported figures are less than 100% due to non-request related web server activity such as health monitoring and garbage collection, and also the difficulty of attributing all kernel mode CPU usage correctly with the current kernel instrumentation.

The final column in Table 2 labelled “*Model Error*”, records how the resource consumption of the constructed model differs from that actually measured. This figure is computed as the percentage difference in resource consumption between the requests as parsed, and a synthetic workload generated by mixing the cluster representatives in proportion to their cluster sizes. In all cases, the cluster sizes and representatives can be used to concisely represent the experimental workload to within 3.2%.

As a useful cross-check, from experiment 1.1 we have a centroid that represents the resource usage and control flow of requests of type A, measured in isolation; from experiment 1.2 we have a similar centroid for requests of type B. Table 2 shows that experiment 1.5 contains requests with a 266/234 mix of type A and B requests, so we can compare the CPU time consumed by all requests in experiment 1.5, and the CPU time predicted by 266 type A centroids added to 234 type B centroids. The prediction falls short of the observed value by just 3.5%, presumably due to worse cache locality. Repeating this computation for experiment 1.6, the deficit is 3.4%.

6.1.2 Network activity

Table 3 shows the results obtained using workloads based on request types C and D, which differ in consumption of network bandwidth. The *Accntd CPU %* figures are noticeably lower for these experiments. However this is not surprising since it is common when load increases for multiple incoming network packets to be processed in the same deferred procedure call. Although 100% of the network packets are correctly ascribed to requests, there

Internal concurrency (1)							
Exp No	Workload / #Clients	Accntd CPU %	Clusters Found	Dia.	Min. Sep.	Model Error	
Single	3.1 500E x1	96.6%	500	0.7	0.0	1.2%	
	3.2 500E x5	97.1%	493	0.9	7.7	0.6%	
			2	0.3	8.5		
			2 †	0.7	130		
			1	0.0	101		
Mixed	3.3 500(B/E) x1	95.8%	267E	0.5	8.7	0.6%	
			232B	0.4	8.8		
			1E	0.0	34.7		
	3.4 500(B/E) x5	95.8%	249B	0.7	8.8	0.4%	
			244E	1.0	6.3		
Single	3.5 500 (A/B/E) x1	96.1%	172E	0.5	8.9	3.0%	
			168A	0.6	8.1		
			160B	0.5	8.1		
	3.6 500 (A/B/E) x5	96.2%	187E	0.9	0.0	0.3%	
			152B	0.6	8.6		
Mixed			160A	1.2	8.6		
			1A †	0.0	130		

Table 4: Clusters produced from single and mixed request type workloads consuming CPU only, where the request contains internal concurrency, for both concurrent and serialized request invocations. See the Table 2 caption for explanation of the column headings.

are places where insufficient instrumentation is present to safely account computation to an individual request. In these cases, the parser errs on the conservative side.

6.2 Concurrency and internal structure

Figure 11 shows canonical versions of the four compute bound requests of Table 1 with very different internal structure. The first two requests (A and B) perform sequential computations of different lengths, the third (E) performs the same amount of work as B, but in two parallel threads. The fourth request (F) also consumes the same amount of resource as B, but this time using a synchronous RPC-style interaction with a second worker thread.

Whilst three of these requests consume exactly the same amount of CPU resource, they would have significantly different behaviour on a multiprocessor machine from a response time or latency perspective. When extracting workload models, we believe it is important to capture these differences in concurrency structure.

Tables 4 and 5 show the clustering results of a suite of experiments constructed using various combinations of the requests described above. From the tables, it is clear that our distance metric and clustering algorithm are capable of separating requests that differ only in internal concurrency structure. Note in particular, experiment 4.8 of Table 5, where the 4 request types fall into 4 well separated clusters, with just 2 outliers.

Internal concurrency (2)							
Exp No	Workload / #Clients	Accntd CPU %	Clusters Found	Dia.	Min. Sep.	Model Error	
Single	4.1 500F x1	96.8%	499	1.7	7.3	4.1%	
			1	0.0	7.3		
	4.2 500F x5	96.5%	496	1.3	6.4	1.2%	
			2	0.6	6.4		
			1	0.0	9.3		
Mixed	4.3 500(A/F) x1	96.7%	248F	1.6	19.2	.01%	
			252A	0.4	19.2		
	4.4 500(A/F) x5	96.6%	236F	1.3	9.4	0.4%	
			262A	0.7	16.9		
			1F †	0.0	131		
Single	4.5 500 (A/E/F) x1	96.8%	192F	1.8	7.9	0.2%	
			169A	0.9	17.6		
			139E	0.5	8.1		
	4.6 500 (A/E/F) x5	96.6%	195F	1.3	6.9	0.9%	
			144E	1.2	9.1		
Mixed			155A	0.7	9.2		
			3F †	1.2	131		
			1E	0.0	9.2		
			2F	2.0	6.9		
	4.7 500 (A/B/E/F) x1	96.3%	133F	1.9	5.5	1.9%	
Single			129A	0.8	7.4		
			120E	0.9	7.8		
			117B	0.1	7.4		
			1F	0.0	5.5		
	4.8 500 (A/B/E/F) x5	96.3%	132B	1.2	7.7	4.2%	
Mixed			131F	1.2	8.0		
			119A	0.5	8.0		
			116E	0.4	5.5		
			1A	0.0	5.5		
			1F	0.0	8.3		

Table 5: Clusters produced from single and mixed request type workloads consuming CPU only, where the request contains internal concurrency and blocking, for both concurrent and serialized request invocations. See the Table 2 caption for explanation of the column headings.

6.3 An anomaly detection example

In several of the above experiments, we noticed occasional unexpected outlier requests, which were always placed in a cluster on their own (marked with a † in result tables). Examination of these individual requests revealed that in every case a 100ms chunk of CPU was being consumed inside a DPC. Using sampling profiler events logged by ETW during the offending intervals the problem was tracked down to a 3Com miniport Ethernet driver calling `KeDelayExecution(100000)`³ from its transmit callback function!

The above example gives some concrete proof that Magpie can highlight apparently anomalous behaviour using extracted requests. Other common causes of outlier requests include JIT compilation, loading shared libraries, cache misses and genuine performance anomalies. Being able to identify and isolate these outliers is an advantage for accurate workload modelling.

³This function is implemented as a busy wait and the documentation clearly states it should not be used for delays of more than 100us.

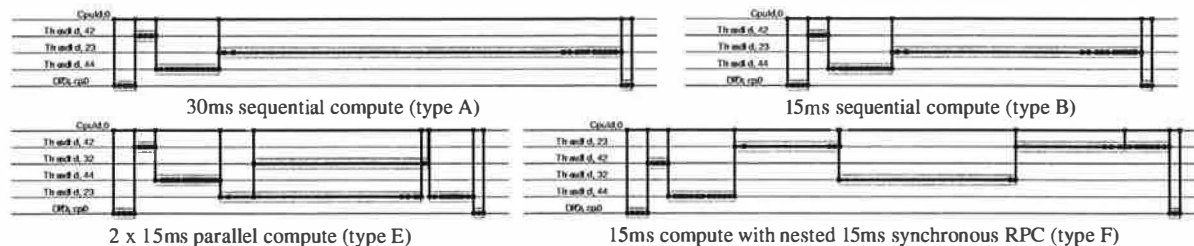


Figure 11: Canonical versions of four compute-bound HTTP requests with different internal concurrency structure.

7 Evaluation

We now turn to an evaluation of the toolchain with more realistic scenarios: a small two-tier web site and an enterprise-class database server.

7.1 Duwamish

In this section we extract requests from a distributed system and look at the accuracy of the derived workload model. The experimental setup is a two machine system running the Duwamish bookstore, a sample e-commerce application that is distributed with Visual Studio. We augmented the Duwamish database and the images stored at the web server with auto-generated data in accordance with the volume and size distributions mandated by TPC-W [21].

As in previous experiments, we first obtained results when all requests have the same URL, and then looked at the clusters produced from a mixed workload. Three dynamic HTML URLs were used, each with different characteristics:

1. The *book.aspx* page invokes a single database query to retrieve a book item and look up its associated data such as author and publisher. An image of the book cover may also be retrieved from the web server file system.
2. The *categories.aspx* page retrieves the details associated with the specified category by making three RPCs to the database.
3. The *logon.aspx* URL runs only on the web server.

As described in Section 4, a stitcher matches packet events within request fragments from individual machines to produce a single canonical form of the request across multiple machines. Table 6 shows the results of clustering on the WEB and SQL fragments alone, as well as on the entire request. For clarity, we have reported just the maximum cluster diameter and the minimum inter-cluster separation of each set of clusters. Similar to previous experiments, we report the fraction of relevant CPU that was included in the extracted requests in the “*Accntd CPU%*” column.

Exp No	Workload	Accntd CPU %	Clusters Found	Max. Dia.	Min. Sep.
5.1	Book(B)	82.8%	WEB: 404+92+4	0.8	3.6
		96.1%	SQL: 500	0.5	0.0
		-	E2E: 408+92	1.2	5.1
5.2	Logon(L)	86.5%	WEB: 497+2+1	1.2	7.3
5.3	Categories (C)	95.2%	WEB: 445+26+29	2.3	14.9
		97.3%	SQL: 1499+1	1.3	16.4
		-	E2E: 444+26+29+1	2.6	17.9
5.4	Mixed (B/L/C)	91.7%	WEB: (138L+1B)+220B+61C+31C+24C+10*25C	4.9	5.9
		96.7%	SQL: (141C ₁ +122C ₂ +141C ₃ +221B)+9C ₂ +10C ₂	2.3	19.7
		-	E2E: (138L+1B)+220B+55C+47C+14C+10*25C	5.1	8.2

Table 6: Clusters found from Duwamish requests, with both single and mixed URL workloads. Results are shown from clustering the workloads from individual machines (WEB and SQL) as well as “end-to-end requests” across both (E2E).

Closer inspection of the resulting clusters reveals that the *book* requests are primarily differentiated by whether a disk read is performed on the web server (to fetch an image of the book cover). On the *categories* page, the amount of network traffic varies between categories and hence one major and two minor clusters are formed. The three stored procedures invoked by *categories.aspx*—*GetCategories()*, *GetBooksByCategoryId()* and *GetDailyPickBooksByCategoryId()*—are identified in the table as *C*₁, *C*₂ and *C*₃ respectively. All of these database request fragments bar one for this URL are put in the same cluster. According to the SQL Server Query Analyzer tool, the stored procedures are all of similar cost, so this is not surprising. The clusters for the mixed workload show that the *book* and *logon* pages form tighter clusters than the *categories* requests, which are spread across several clusters. These results indicate that a workload model based on per-URL measurements will be less representative than one constructed by grouping similar requests according to their observed behaviour.

TPC-C benchmark

#	Size	Contents	$d(0)$	$Dia.$	$Min. Sep.$
1	751	620B+100F+30D+1A	54.30	0.025	9.264
2	392	329A+56E+7D	105.05	0.119	14.393
3	302	266A+30D+3B+3E	29.16	0.116	9.264
4	30	30C	555.45	9.596	81.251
5	21	21C	111.03	4.870	78.080

Key:	A neworder	B payment	C stocklevel
	D orderstatus	E delivery	F version

Table 7: Clusters formed from TPC-C workload. The workload is a constrained ratio mix of 6 different transaction types shown in the key. The additional column $d(0)$ shows the distance of each cluster centroid from the null request and gives an indication of the amount of resource consumption (largely disk I/O in this case).

7.2 TPC-C

The TPC-C benchmark [20] results presented in this section were generated using the Microsoft TPC-C Benchmark Kit v4.51 configured with 10 simultaneous clients and a single server. The resulting database would normally fit entirely in the memory of a modern machine. We therefore ran SQL Server with restricted memory to more accurately reflect the cache behaviour of a realistically dimensioned TPC-C environment.

The clustering algorithm created 5 clusters from 1496 requests. The clusters are quite tightly formed (they have low intra-cluster distances, $Dia.$) and well separated (they have high inter-cluster distances, $Sep.$). Although clusters 4 and 5 have somewhat higher intra-cluster distances, they are so well separated from any other cluster that this is unimportant.

Examining the make-up of the clusters reveals that the amount of I/O performed is the dominant factor in deciding the cluster for a request. Cluster 1 contains all *version* transactions and 99% of *payment* transactions, none of which have any I/O. Cluster 2 contains 95% of *delivery* transactions and 55% of *neworder* transactions: these are the transactions with a small amount of I/O. Cluster 3 holds the remaining 45% of *neworder* transactions, all of which have a moderate amount of I/O. The *orderstatus* transactions are split 45%/10%/45% between clusters 1–3 based on the I/O they contain. Finally, clusters 4 and 5 contain all the *stocklevel* transactions, predicted to be nearly 3 orders of magnitude more expensive than the next most expensive transaction by SQL Server Query Analyzer.

7.2.1 Shared buffer cache resources

Although the above clusters represent a reasonable summary of the benchmark workload in the experimental configuration, they also expose an area requiring further attention. In many applications, and especially in database servers, a shared buffer cache is the dominant factor affecting performance. Our instrumentation does

not yet record cache and memory references, observing only the disk I/O associated with cache misses and log writes. Given the explicit SQL buffer cache API it would be a simple matter to record the locality and sequence of pages and tables referenced by each query. We believe that this extra information will better distinguish between transaction types and may allow us to predict miss rates with different cache sizes as described in [15], but this remains an area for future work.

8 Related work

The most closely related work to Magpie is Pinpoint [8]. Pinpoint collects end-to-end traces of client requests in a J2EE environment by tagging each call with a request ID. This is simpler than using event correlation to extract requests, but requires propagation of a global request ID, which is not always possible with heterogeneous software components. The aim of Pinpoint is to diagnose faults by applying statistical methods to identify components that are highly correlated with failed requests. This is in contrast to the Magpie goal of recording not only the path of each request, but also its resource consumption, and hence being able to understand and model system performance.

Aguilera et al. [1] have proposed statistical methods to derive causal paths in a distributed system from traces of communications. Their approach is minimally invasive, requiring no tracing support above the RPC layer. However, by treating each machine as a black box, they sacrifice the ability to separate out interleaved requests on a single machine, and thus cannot attribute CPU and disk usage accurately. The approach is aimed at examining statistically common causal paths to find sources of high latency. Magpie's request parsing on the other hand captures all causal paths in a workload, including relatively rare (but possibly anomalous) ones.

Distributed event-based monitors and debuggers [2, 4, 13] track event sequences across machines, but do not monitor resource usage, which is essential for performance analysis. Conversely, many systems track request latency on a single system but do not address the distributed case. TIPME [9] tracked the latency of interactive operations initiated by input to the X Window System. Whole Path Profiling [14] traces the control flow patterns between basic blocks in a running program.

Similar approaches on different operating systems include the Linux Trace Toolkit [22], which tracks request latency on a single machine. The Magpie toolchain could easily be built to consume LTT events instead of ETW events. A more sophisticated instrumentation framework is Solaris DTrace [6], which allows arbitrary predicates and actions to be associated with instrumentation points. DTrace provides an option for speculative tracing, which

could potentially be a lightweight mechanism for enabling request sampling.

Chen and Perkowitz [7] measure web application response times by embedding JavaScript in the web pages being fetched, i.e. by modifying the content being served rather than instrumenting client or server code. The aggregated data gives a view of client-side latency that would complement the detailed server-side workload characterisation obtained using Magpie.

9 Conclusion

In this paper we described the Magpie toolchain that takes stand-alone events generated by operating system, middleware and application components, correlates related events to extract individual requests, expresses those requests in a canonicalized form and then finally clusters them to produce a workload model. We validated our approach against traces of synthetic workloads, and showed that our approach is promising for more complicated applications.

We have shown that by using Magpie to isolate the resource demands and the path taken by requests, we can construct stochastic models that give a good representation of a workload's behaviour. A great advantage of Magpie is that these request structures are learnt by observing the live system under a realistic workload. As a consequence, the parsed event trace of each individual request is recorded, giving a detailed picture of how requests are actually being serviced within the system.

Acknowledgements

We gratefully acknowledge the encouragement and insightful comments of our shepherd Eric Brewer, and many proof-readers especially Steve Hand, Tim Harris and Andrew Herbert. Thanks also to Dushyanth Narayanan and James Bulpin for past contributions to the Magpie project.

References

- [1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 74–89, Oct. 2003.
- [2] E. Al-Shaer, H. Abdel-Wahab, and K. Maly. HiFi: A new monitoring architecture for distributed systems management. In *Proc. IEEE 19th International Conference on Distributed Computing Systems (ICDCS'99)*, pages 171–178, May 1999.
- [3] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: on-line modelling and performance-aware systems. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, pages 85–90, May 2003.
- [4] P. C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Transactions on Computer Systems (TOCS)*, 13(1):1–31, 1995.
- [5] H. Bunke. Recent developments in graph matching. In *Proc. 15th International Conference on Pattern Recognition*, pages 117–124, 2000.
- [6] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proc. USENIX Annual Technical Conference*, pages 15–28, June 2004.
- [7] J. B. Chen and M. Perkowitz. Using end-user latency to manage internet infrastructure. In *Proc. 2nd Workshop on Industrial Experiences with Systems Software WIESS'02*, Dec. 2002.
- [8] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *Proc. 1st Symposium on Networked Systems Design and Implementation (NSDI'04)*, pages 309–322, Mar. 2004.
- [9] Y. Endo and M. Seltzer. Improving interactive performance using TIPME. In *Proc. ACM SIGMETRICS*, June 2000.
- [10] D. Gao, C. S. Jensen, R. T. Snodgrass, and M. D. Soo. Join operations in temporal databases. Technical Report TR-71, TIME-CENTER, Oct. 2002.
- [11] R. Isaacs, P. Barham, J. Bulpin, R. Mortier, and D. Narayanan. Request extraction in Magpie: events, schemas and temporal joins. In *11th ACM SIGOPS European Workshop*, Sept. 2004.
- [12] J.O.Kephart and D.M.Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, Jan. 2003.
- [13] J. Joyce, G. Lomow, K. Slind, and B. Unger. Monitoring distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 5(2):121–150, 1987.
- [14] J. R. Larus. Whole program paths. In *Proc. ACM conference on Programming Language Design and Implementation (SIGPLAN'99)*, pages 259–269, June 1999.
- [15] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.
- [16] Microsoft Application Center Test 1.0, Visual Studio .NET Edition. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/act/html/actml_main.asp, May 2004.
- [17] Microsoft Corp. Event Tracing for Windows (ETW). http://msdn.microsoft.com/library/en-us/perfmon/base/event_tracing.asp, 2002.
- [18] I. Park and M. K. Raghuraman. Server diagnosis using request tracking. In *1st Workshop on the Design of Self-Managing Systems, held in conjunction with DSN 2003*, June 2003.
- [19] F. Risso and L. Degioanni. An architecture for high performance network analysis. In *Proc. 6th IEEE Symposium on Computers and Communications*, pages 686–693, July 2001.
- [20] Transaction Processing Performance Council. *TPC Benchmark C (On-line Transaction Processing) Specification*. <http://www.tpc.org/tpcc/>.
- [21] Transaction Processing Performance Council. *TPC Benchmark W (Web Commerce) Specification*. <http://www.tpc.org/tpcw/>.
- [22] K. Yaghmour and M. R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *Proc. USENIX Annual Technical Conference*, June 2000.

Using Model Checking to Find Serious File System Errors

Junfeng Yang, Paul Twohey, Dawson Engler *
{junfeng, twohey, engler}@cs.stanford.edu
Computer Systems Laboratory
Stanford University
Stanford, CA 94305, U.S.A.

Madanlal Musuvathi
madanm@microsoft.com
Microsoft Research
One Microsoft Way
Redmond, WA 98052, U.S.A.

Abstract

This paper shows how to use model checking to find serious errors in file systems. Model checking is a formal verification technique tuned for finding corner-case errors by comprehensively exploring the state spaces defined by a system. File systems have two dynamics that make them attractive for such an approach. First, their errors are some of the most serious, since they can destroy persistent data and lead to unrecoverable corruption. Second, traditional testing needs an impractical, exponential number of test cases to check that the system will recover if it crashes at any point during execution. Model checking employs a variety of state-reducing techniques that allow it to explore such vast state spaces efficiently.

We built a system, FiSC, for model checking file systems. We applied it to three widely-used, heavily-tested file systems: ext3 [13], JFS [21], and ReiserFS [27]. We found serious bugs in all of them, 32 in total. Most have led to patches within a day of diagnosis. For each file system, FiSC found demonstrable events leading to the unrecoverable destruction of metadata and entire directories, including the file system root directory “/”.

1 Introduction

File system errors are some of the most destructive errors possible. Since almost all deployed file systems reside in the operating system kernel, even a simple error can crash the entire system, most likely in the midst of a mutation to stable state. Bugs in file system code can range from those that cause “mere” reboots to those that lead to unrecoverable errors in stable on disk state. In such cases, mindlessly rebooting the machine will not correct or mask the errors and, in fact, can make the situation worse.

*This research was supported by NSF grant CCR-0326227 and DARPA grant F29601-03-2-0117. Dawson Engler is partially supported by Coverity and an NSF Career award.

Not only are errors in file systems dangerous, file system code is simultaneously both difficult to reason about and difficult to test. The file system must correctly recover to an internally consistent state if the system crashes at *any* point, regardless of what data is being mutated, flushed or not flushed to disk, and what invariants have been violated. Anticipating all possible failures and correctly recovering from them is known to be hard; our results do not contradict this perception.

The importance of file system errors has led to the development of many file system stress test frameworks; two good ones are [24, 30]. However, these focus mostly on non-crash based errors such as checking that the file system operations create, delete and link objects correctly. Testing that a file system correctly recovers from a crash requires doing reconstruction and then comparing the reconstructed state to a known legal state. The cost of a single crash-reboot-reconstruct cycle (typically a minute or more) makes it impossible to test more than a tiny fraction of the exponential number of crash possibilities. Consequently, just when implementors need validation the most, testing is least effective. Thus, even heavily-tested systems have errors that only arise after they are deployed, making their errors all but impossible to eliminate or even replicate.

In this paper, we use model checking to systematically test and find errors in file systems. Model checking [5, 19, 22] is a formal verification technique that systematically enumerates the possible states of a system by exploring the nondeterministic events in the system. Model checkers employ various *state reduction* techniques to efficiently explore the resulting exponential state space. For instance, generated states can be stored in a hash table to avoid redundantly exploring the same state. Also, by inspecting the system state, model checkers can identify similar set of states and prioritize the search towards previously unexplored behaviors in the system. When applicable, such a systematic exploration can achieve the effect of impractically massive testing by avoiding the

redundancy that would occur in conventional testing.

The dominant cost of traditional model checking is the effort needed to write an abstract specification of the system (commonly referred to as the “model”). This upfront cost has traditionally made model checking completely impractical for large systems. A sufficiently detailed model can be as large as the checked system. Empirically, implementors often refuse to write them; those that are written have errors and, even if they do not, they “drift” as the implementation is modified but the model is not [6].

Recent work has developed *implementation-level* model checkers that check implementation code directly without requiring an abstract specification [18, 25, 26]. We leverage this approach to create a model checking infrastructure, the File System Checker (FiSC), which lets implementors model-check real, unmodified file systems with relatively little model checking knowledge. FiSC is built on CMC, an explicit state space, implementation model checker we developed in previous work [25, 26], which lets us run an entire operating system inside of the model checker. This allows us to check a file system *in situ* rather than attempting the difficult task of extracting it from the operating system kernel.

We applied FiSC to three widely-used, heavily-tested file systems, JFS [21], ReiserFS [27], and ext3 [13]. We found serious bugs in all of them, 32 in total. Most have led to patches within a day of diagnosis. For each file system, FiSC found demonstrable events leading to the unrecoverable destruction of metadata and entire directories, including the file system root directory “/”.

The rest of the paper is as follows. We give an overview of both FiSC (§2) and how to check a file system with it (§3). We then describe: the checks FiSC performs (§4), the optimizations it does (§5), and how it checks file system recovery code (§6). We then discuss results (§7) and our experiences using FiSC (§8), including sources of false positives and false negatives. We then conclude.

2 Checking Overview

Our system is comprised of four parts: (1) CMC, an explicit state model checker running the Linux kernel, (2) a file system test driver, (3) a permutation checker which verifies that a file system can recover no matter what order buffer cache contents are written to disk, and (4) a *fsck* recovery checker. The model checker starts in an initial pristine state (an empty, formatted disk) and recursively generates and checks successive states by systematically executing state transitions. Transitions are either test driver operations or FS-specific kernel threads which flush blocks to disk. The test driver is conceptually similar to a program run during testing. It creates, removes, and renames files, directories, and hard links;

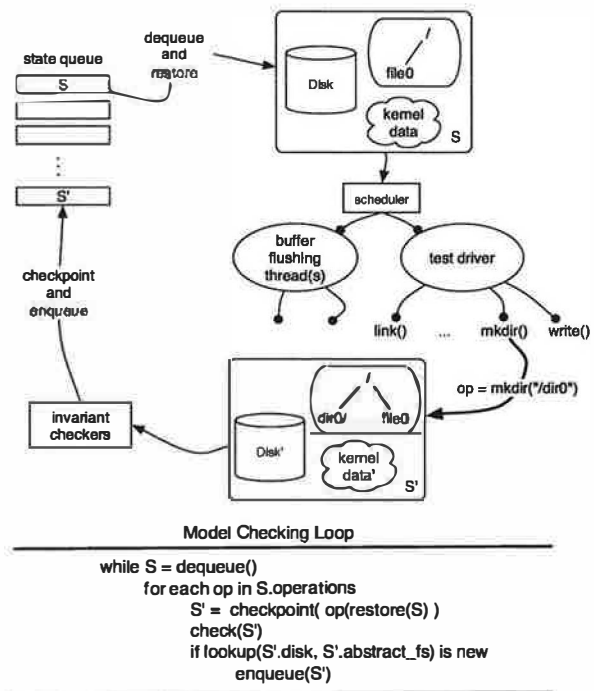


Figure 1: State exploration and checking overview. FiSC’s main loop picks a state S from the state queue and then iteratively generates its successor states by applying each possible operation to a restored copy of S . The generated state S' is checked for validity and, if valid and not explored before, inserted onto the state queue.

writes to and truncates files; and mounts and unmounts the file system. Figure 1 shows this process.

As each new state is generated, we intercept all disk writes done by the checked file system and forward them to the permutation checker, which checks that the disk is in a state that *fsck* can repair to produce a valid file system after each subset of all possible disk writes. This avoids storing a separate state for each permutation and allows FiSC to choose which permutations to check. This checker is explained in Section 4.2. We run *fsck* on the *host* system outside of the model checker and use a small shared library to capture all the disk accesses *fsck* makes while repairing the file system generated by writing a permutation. We feed these *fsck* generated writes into the crash recovery checker. This checker allows FiSC to recursively check for failures in *fsck* and is covered in Section 6.

Figure 2 outlines the operation of the permutation and *fsck* recovery checkers. Both checkers copy the disk from the starting state of a transition and write onto the copy to avoid perturbing the system. After the copied disk is modified the model checker traverses its file system, recording the properties it checks for consistency in a model of the file system. Currently these are the name,

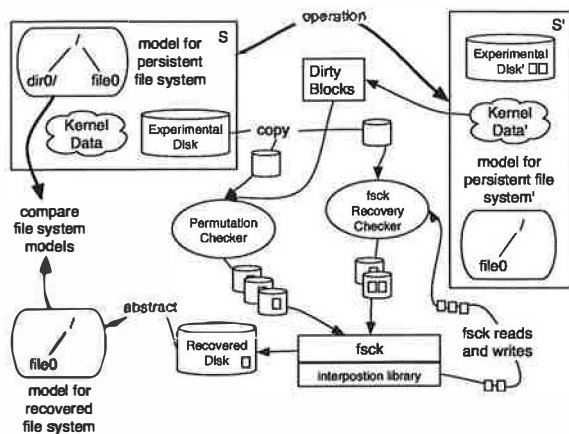


Figure 2: Disk permutation and `fsck` recovery checkers.

size, and link count of every file and directory in the system along with the contents of each directory. Note that this is a model of file system *data*, not file system *code*. The code to traverse, create, and manipulate the file system model mirrors the system call interface and can be reused to check many file systems. We check that this model matches one of the possible valid file systems, which are computed as in Section 3.2. An error is flagged in case of a mismatch.

After each new state is generated our system runs a series of invariant checkers looking for file system errors. If an error is found FiSC (1) emits an error message, (2) stores a trace for later error diagnosis that records all the nondeterministic choices it made to get to the error, and (3) discards the state. If there is no error, FiSC looks at the new state and puts it on the state queue if it has not already visited a similar state (§5.1). Otherwise, it discards the state.

New states are *checkpointed* and added to the state queue for later exploration. Checkpointing the kernel state captures the current execution environment so that it can be put on the state queue and *restored* later when the model checker decides to take it off the state queue and explore its operations. This state consists of the kernel heap and data, the disk, an abstract model of the current file system, and additional data necessary for invariant checks. As discussed in Section 5.1, FiSC searches the state space using breadth- or depth-first search along with some simple heuristics.

2.1 The Checking Environment

Similar to unit testing, model-checking a file system requires selecting two layers at which to cut the checked system. One layer defines the external interface that the test driver runs on. In our case we have the driver run

atop the system call interface. The other layer provides a “fake environment” that the checked system runs on. We need this *environment model* because the checked file system does not run on bare hardware. Instead, FiSC provides a virtual block device that models a disk as a collection of sectors that can be written atomically. The block device driver layer is a natural place to cut as it is the only relatively well-documented boundary between in-core and persistent data.

Modern Unix derivatives provide a Virtual File System (VFS) interface [28]. While the VFS seems like a good place to cut, it varies significantly across operating systems and even across different versions of the same kernel. Further it has many functions with subtle dependencies. By instead cutting along the system call layer we avoid the headache of modeling these sparsely documented interactions. We also make our system easier to port and are able to check the VFS implementation, a design decision validated by the two bugs we found in the VFS code (§7).

3 Checking a New File System

This section gives an overview of what a file system implementor must do to check a new file system.

3.1 Basic Setup

Because CMC encases Linux, a file system that already runs within Linux and conforms to FiSC’s assumptions of file system behavior will require relatively few modifications before it can be checked.

FiSC needs to know the minimum disk and memory sizes the file system requires. Ext3 had the smallest requirements: a 2MB disk and 16 pages of memory. ReiserFS had the highest: a 30MB disk and 128 pages. In addition, it needs the commands to make and recover the file system (usually with `mkfs` and `fsck` respectively). Ideally, the developer provides three different `fsck` options for: (1) default recovery, (2) “slow” full recovery, and (3) “fast” recovery that only replays the journal so that the three recovery modes may be checked against each other (§4).

In addition to providing these facts, an implementor may have to modify their file system to expose dirty blocks. Some consistency checks require knowing which buffers are dirty (§4.2). A file system, like ReiserFS, that uses its own machinery for tracking dirty buffers must be changed to explicitly indicate such dirty buffers.

When a file system fits within FiSC’s model of how a file system works (as do ext3 and JFS) it takes a few days to start checking. On the other hand, ReiserFS took between one and two weeks of effort to run in FiSC as it violated one of the larger assumptions we made. As stated earlier, during crash checking FiSC mounts a copy of the disk used by the checked file system as a second block

device that it uses to check the original. Thus, the file system must independently manage two disks in a reentrant manner. Unfortunately, ReiserFS does not do so: it uses a single kernel thread to perform journal writes for all mounted devices, which causes a deadlock when the journal thread writes to the log, FiSC suspends it, creates a copy of the disk, and then remounts the file system. Remounting normally replays the journal, but this requires writing to the journal – which deadlocks waiting for the suspended journal thread to run. We fixed the problem by modifying ReiserFS to not wake the journal thread when a clean file system is mounted read-only.

3.2 Modeling the File System

After every file system operation, FiSC compares the checked file system against what it believes is the correct *volatile file system* (VolatileFS). The VolatileFS reflects the effects of all file system operations done sequentially up through the last one. Because it is defined by various standards rather than being FS-specific, FiSC can construct it as follows. After FiSC performs an operation (e.g., `mkdir`, `link`) to the checked concrete system, it also emulates the operation's effect on a "fake" abstract file system. It then verifies that the checked and abstract file systems are equivalent using a lossy comparison that discards details such as time.

After every disk write, FiSC compares the checked file system against a model of what it believes to be the current *stable file system* (StableFS). The StableFS reflects the state the file system should recover to after a crash. At any point, running a file system's `fsck` repair utility on the current disk should always produce a file system equivalent to this StableFS.

Unlike the VolatileFS, the StableFS is FS-specific. Different file systems make wildly different guarantees as to what will be recovered after a crash. The ext2 [13] file system provides almost none, a journaling file system typically intends to recover up to the last completed log record or commit point, and a soft-updates [17] file system recovers to a difficult-to-specify mix of old and new data.

Determining how the StableFS evolves requires determining two FS-specific facts: (1) when it can legally change and (2) what it changes to. FiSC requires that the implementor modify the checked file system to call into the model checker to indicate when the StableFS changes. For journaling file systems this change typically occurs when a journal commit record is written to disk. We were able to identify and annotate the commit records relatively easily for ext3 and ReiserFS. JFS was more difficult. In the end, after a variety of false starts, we gave up trying to determine which journal write represented a commit-point and instead let the StableFS change after *any* journal write. We assume a file system

implementor could do a better job.

Once we know that the StableFS changes, we need to know what it changes to. Doing so is difficult since it essentially requires writing a crash recovery specification for each file system. While we assume a file system implementor could do so, we check systems we did not build. Thus, we take a shortcut and use `fsck` to generate the StableFS for us. We copy the experimental disk, run `fsck` to reconstruct a file system image after the committed operations, and traverse the file system, recording properties of interest. This approach can miss errors since we have no guarantee that `fsck` will produce the correct state. However, it is relatively unlikely that `fsck` will fail when repairing a perfectly behaving disk. It is even more unlikely that if it does fail that it will do so in the same way for the many subsequent crashed disks to which the persistent file system model will be compared.

3.3 Checking More Thoroughly

Once a basic file system is up and being checked, there are three main strategies an implementor can follow to check their file system more thoroughly: downscaling [10], canonicalization, and exposing choice points. We talk about each below.

Downscale. Operationally this means making everything as small as plausible. Caches become one or two entries large, file systems just a few "nodes" (where a node is a file or directory). Model checking works best at ferreting out complex interactions of a small number of nouns (files, directories, blocks, threads, etc) since this small number allows caching techniques to give the most leverage. There were three main places we downscaled. First, making disk small (megabytes rather than gigabytes). Second, checking small file system topologies, typically 2-4 nodes. Finally, reducing the size of "virtual memory" of the checked Linux system to a small number of pages.

Canonicalization. This technique modifies states so that state hashing will not see "irrelevant" differences. In practice, the most common canonicalization is to set as many things as possible to constant values: clearing inode generation numbers, mount counts, time fields; zeroing freed memory and unused disk blocks (especially journal blocks).

Many canonicalizations require FS-specific knowledge and thus must be done by the implementor. However, FiSC does do two generic canonicalization. First, it constrains the search space by only writing two different values to data blocks, significantly reducing the number of states while still providing enough resolution to catch data errors. Second, before hashing a model of a file system, FiSC transforms the file system to remove superficial differences, by renaming files and directories so that there is always a sequential numbering among file

system objects. For example a file system with one directory and three files “a,” “b,” and “c” will have the same model as another file system with one directory and three files “1,” “2,” and “3” if the files have the same length and content. Canonicalization lets us move our search space away from searching for rare filename-specific bugs and toward the relatively more common bugs that arise while creating many file system topologies.

Expose choice points. Making sources of nondeterminism (“choice points”) visible to FiSC lets it search the set of possible file system behaviors more thoroughly. A low-level example is adding code to fail FS-specific allocators. More generally, whenever a file system makes a decision based on an arbitrary time constraint or environmental feature, we change it to call into FiSC so that FiSC can choose to explore each possible decision in every state that reaches that point.

Mechanically, exposing a choice point reduces to modifying the file system code to call “choose(*n*)” where *n* is the number of possible decision alternatives. choose will appear to return to this callsite *n* times, with the return values 0, ..., (*n* - 1). The caller uses this return value to pick which of the *n* possible actions to perform. An example: both ReiserFS and ext3 flush their in-memory journals to disk after a given amount of time has lapsed. We replaced this time check with a call to choose(2) and modified the caller so that when choose returns 0 the code flushes the commit record; when it returns 1 it does not. As another example, file systems check the buffer cache before issuing disk reads. Without care, this means that the “cache miss” path will rarely be checked (especially since we check tiny file system topologies). We solve this problem by using choose on the success path of the buffer cache read routine to ensure FiSC also explores the miss path. In addition, FiSC generically fails memory allocation routines and permission checks.

When inserting choice points, the implementor can exploit well-defined internal interfaces to increase the set of explored actions. Interface specifications typically allow a range of actions, of which an implementation will pick some subset. For example, many routines specify that any invocation may return an “out of memory” error. However their actual implementation may only allocate memory on certain paths, or perhaps never do any allocations at all. It is a mistake to only fail the specific allocation calls an implementation performs since this almost certainly means that many callers and system configurations will never see such failures. The simple fix is to insert a choice point as the routine’s first action allowing the model checker to test that failure is handled on each call.

Unfortunately, it is not always easy to expose choice points and may require restructuring parts of the system

to remove artificial constraints. The most invasive example of these modifications are the changes to the buffer cache we made so that the permutation checker (§4.2) would be able to see all possible buffer write orderings.

4 Checkers

This section describes the checks FiSC performs.

4.1 Generic Checks

FiSC inspects the actual state of the system and can thus catch errors that are difficult or impossible to diagnose with static analysis. It is capable of doing a set of general checks that could apply to any code run in the kernel:

Deadlock. We instrument the lock acquisition and release routines to check for circular waits.

NULL. FiSC reports an error whenever the kernel dereferences a NULL pointer.

Paired functions. There are some kernel functions, like `iget`, `iput` for inode allocation and `dget`, `dput` for directory cache entries, which should always be called in pairs. We instrument these functions in the kernel and then check that they are always called in pairs while running the model checker.

Memory leak. We instrument the memory allocation and deallocation functions so FiSC can track currently used memory. We also altered the system-wide freelist to prevent memory consumers from allocating objects without the model checker’s knowledge. After every state transition we stop the system and perform a conservative traversal [2] of the stack and the heap looking for allocated memory with no references.

No silent failures. The kernel does not request a resource for which it does not have a specific use planned. Thus, it is likely a bug when a system call returns success after it calls a resource allocation routine that fails. The exception to this pattern is when code loops until it acquires a resource. In which case, we generate a false positive when a function fails during the first iteration of the loop but later succeeds. We suppress these false positives by manually marking functions with resource acquisition loops.

4.2 Consistency Checks

FiSC checks the following consistency properties.

System calls map to actions. A mutation of the file system that indicates success (usually a system call with a return value of zero) should produce a user-visible change, while an indication of failure should produce no such change. We use a reference model (the VolatileFS) to ensure that when an operation produces a user-visible change it is the correct change.

Recoverable disk write ordering. As described in §2, we write arbitrary combinations of dirty buffer cache entries to disk, checking that the system recovers to a valid state. File system recovery code typically requires that

disk writes happen in certain stylized orders. Illegal orders may not interfere with normal system operation, but will lead to unrecoverable data loss if a crash occurs at an inopportune moment. Comprehensively checking for these errors requires we (1) have the largest legal set of possible dirty buffers in memory and (2) flush combinations of these blocks to disk at every legal opportunity. Unfortunately, many file systems (all those we check) thwart these desires by using a background thread to periodically write dirty blocks to disk. These cleaned blocks will not be available for subsequent reorder checking, falsely constraining the schedules we can generate. Further, the vagaries of thread scheduling can hide vulnerabilities — if the thread does not run when the system is in a vulnerable state then the dangerous disk writes will not happen. Thus we modified this thread to do nothing and instead have the model checker track all blocks that could be legally written. Whenever a block is added to this set we write out different permutations of the set, and verify that running `fsck` produces a valid file system image. The set of possible blocks that can be written are (1) all dirty buffers in the buffer cache (dirty buffers may be written in any order) and (2) all requests in the disk queue (disks routinely reorder the disk queue).

This set is initially empty. Blocks are added whenever a buffer cache entry is marked dirty. Blocks are removed from this set in four ways: (1) they are deleted from the buffer cache, (2) marked clean, (3) the file system explicitly waits for the block to be written or (4) the file system forces a synchronous write of a specific buffer or the entire disk request queue.

Changed buffers are marked dirty. When a file system changes a block in the buffer cache it needs to mark it as dirty so the operating system knows it should eventually write the block back to disk. Blocks that are not marked as dirty may be flushed from the cache at any time. Initially we thought we could use the generic dirty bit associated with each buffer to track the “dirtiness” of a buffer, but each file system has a slightly different concept of what it means for a buffer to be dirty. For example, `ext3` considers a buffer dirty if one of the following conditions is true: (1) the generic dirty bit is set, (2) the buffer is journaled and the journal dirty bit is set, or (3) the buffer is journaled and it has been revoked and the revocation is valid. Discovering dirty buffer invariants requires intimate knowledge of the file system design; thus we have only run this checker on `ext3`.

Buffer consistency. Each journaling file system associates state with each buffer it uses from the buffer cache and has rules about how that state may change. For example a buffer managed by `ext3` may not be marked both dirty and “journal dirty.” That is, it should be written first to the journal (journal dirty), and then written to the appropriate location on disk (dirty).

Double fsck. By default `fsck` on a journaled file system simply replays the journal. We compare the file system resulting from recovering in this manner with one generated after running `fsck` in a comprehensive mode that scans the entire disk checking for consistency. If they differ, at least one is wrong.

5 Scaling the System

As we brought our system online we ran into a number of performance and memory bottlenecks. This section describes our most important optimizations.

5.1 State Hashing and Search

Exploring an exponential state space is a game where you ignore (hopefully) irrelevant details in a quest to only explore states that differ in non-superficial ways. FiSC plays this game in two places: (1) state hashing, where it selectively discards details to make bit-level different states equivalent and (2) searching, when it picks the next state to explore. We describe both below.

We initially hashed most things in the checked file system’s state, such as the heap, data segment, and the raw disk. In practice this meant it was hard to comprehensively explore “interesting” states since the model checker spent its time re-exploring states that were not that much different from each other. After iterative experimentation we settled on only hashing the `VolatileFS`, the `StableFS`, and the list of currently runnable threads. We ignore the heap, thread stacks, and data segment. Users can optionally hash the actual disk image instead of the more abstract `StableFS` to check at a higher-level of detail.

Despite discarding so much detail we rarely can explore all states. Given the size of each checkpoint (roughly 1-3MB), the state queue holding all “to-be-explored” states consumes all memory long before FiSC can exhaust the search space. We stave this exhaustion off by randomly discarding states from the state queue whenever its size exceeds a user-selected threshold.

We provide two heuristic search strategies as alternatives to vanilla DFS or BFS. The first heuristic attempts to stress a file system’s recovery code by preferentially running states whose disks will likely take the most work to repair after a crash. It crudely does so by tracking how many sectors were written when the state’s parent’s disk was recovered and sorts states accordingly. This approach found a data loss error in JFS that we have not been able to trigger with any other strategy.

The second heuristic tries to quantify how different a given state is from previously explored states using a utility score. A state’s utility score is based on how many times states with the same features have already been explored. Features include: the number of dirty blocks a state has, its abstract file system topology, and whether

States	ext3	ReiserFS	JFS
Total	10800	630	4500
Expanded States	2419	142	905
State Transitions	35978	11009	14387
Time			
with Memoization	650	893	3774
without Memoization	7307	29419	4343

Table 1: The number of states, transitions, and the cost of checking each file system until the point at which FiSC runs out of memory. Times are all in seconds. ReiserFS’s relatively large virtual memory requirements limited FiSC checks to roughly an order of magnitude fewer states than the other systems. `fsck` memoization (described in §5.4) speeds checking of ext3 by a factor of 10, and ReiserFS by a factor of 33.

its parent executed new file system statements. A state’s score is an exponentially-weighted sum of the number of times each feature has been seen.

5.2 Systematically Failing Functions

When a transition (e.g., `mkdir`, `creat`) is executed, it may perform many different calls to functions that can fail such as memory allocation or permission checks (§3.3). Blindly causing all combinations of these functions to fail risks having FiSC explore an exponential number of uninteresting, redundant transitions for each state. Additionally, in many cases FS-implementors are relatively uninterested in “unlikely” failures, for example, those only triggered when both memory allocation failures and a disk read error occurs.

Instead, we use an iterative approach — FiSC will first run a transition with no failures, it will then run it failing only a single callsite until all callsites have been failed, it will then similarly fail two callsites, etc. Users can specify the maximum number of failures that FiSC will explore up to. The default is one failure. This approach will find the smallest possible number of failures needed to trigger an error.

5.3 Efficiently Modeling Large Disks

As Figure 3 shows, naively modeling reasonable-sized disks with one contiguous memory allocation severely limits the number of states our model checker can explore as we quickly exhaust available memory. Changing file system code so that it works with a smaller disk is non-trivial and error prone as the code contains mutually-dependent macros, structures, and functions that all rely on offsets in intricate ways. Instead we efficiently model large disks using hash compaction [31]. We keep a database of disk *chunks*, collections of disk sectors, and their hashes. The disk is thus an array of references to

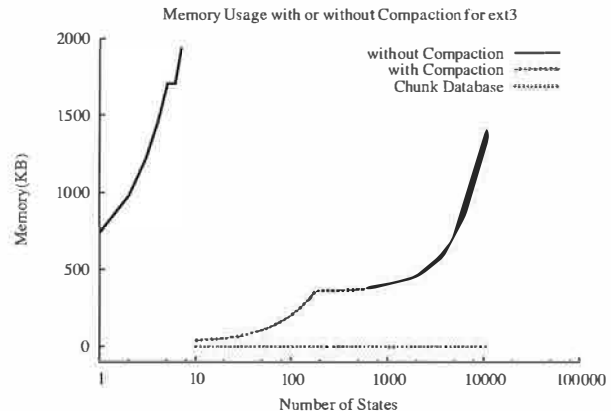


Figure 3: Memory usage when model checking ext3 on a 40MB disk with and without disk compaction. Without compaction the model checker quickly exhausts all the physical memory and dies before it reaches 20 states. The chunk database consumes about 0.2% of the total memory with a maximum of less than 2MB. Note, the spike around the 150 state mark happens because FiSC starts randomly discarding states from its state queue.

hashed chunks. When a write alters a chunk we hash the new value, inserting it into the database if necessary, and have the chunk reference the hash.

5.4 fsck Memoization

Repairing a file system is expensive. It takes about five times as long to run `fsck` as it does to restore a state and generate a new state by executing an operation. If we are not careful, the time to run `fsck` dominates checking. Fortunately, for all practical purposes, recovery code is deterministic: given the same input disk image it should always produce the same output image. This determinism allows us to memoize the result of recovering a specific disk. Before running `fsck` we check if the current disk is in a hash table and, if so, return the already computed result. Otherwise we run `fsck` and add the entry to the table. (As a space optimization we actually just track the sectors read and written by `fsck`.) While memoization is trivial, it gives a huge performance win as seen in Table 1, especially since our `fsck` recovery checker (§6) can run `fsck` 10-20 times after each crash.

5.5 Cluster-based Model Checking

A model checking run makes a set of configuration choices: the number of files and directories to allow, what operations can fail, whether crashes happen during recovery, etc. Exploring different values is expensive, but not doing so can miss bugs. Fortunately, exploration is easily parallelizable. We wrote a script that given a set of configuration settings and remote machines, generates all configurations and remotely executes them.

5.6 Summary

Table 1 shows that FiSC was able to check more than 10k states and more than 35k transitions for ext3 within 650 seconds. The expanded states are those for which all their possible transitions are explored. The data in this section was computed using a Pentium 4 3.2GHz machine with 2GB memory.

6 Crashes During Recovery

A classic recovery mistake is to incorrectly handle a crash that happens during recovery. The number of potential failure scenarios caused by one failure is unwieldy, the number of scenarios caused by a second failure is combinatorially exciting. Unfortunately, since many failures are correlated, such crashes are not uncommon. For example, after a power outage, one might run `fsck` only to have the power go out again while it runs. Similarly, a bad memory board will cause a system crash and then promptly cause another one during recovery.

This section describes how we check that a file system's recovery logic can handle a single crash during recovery. We check that if `fsck` crashes during its first recovery attempt, the final file system (the StableFS) obtained after running `fsck` a second time (on a disk possibly already modified by the previous run) should be the same as if the first attempt succeeded. We do not consider the case where `fsck` crashes repeatedly during recovery. While repeated failure is intellectually interesting, the difficulty in reasoning about errors caused by a single crash is such that implementors have shown a marked disinterest in more elaborate combinations.

Conceptually, the basic algorithm is simple:

1. Given the disk image d_0 after a crash, run `fsck` to completion. We record an ordered "write-list" $WS = (w_1, \dots, w_n)$ of the sectors and values written by `fsck` during recovery. Here w_i is a tuple $\langle s_i, v_i \rangle$, where s_i is the sector written to and v_i is the data written to the sector. In more formal terms, we model `fsck` as a function (denoted $fsck$) that maps from an input disk d to an output disk d' , where the differences between d and d' are the values in the write-set WS . For our purposes these writes are the only effects that running `fsck` has. Moreover, we denote the partial evaluation of $fsck(d)$ after performing writes w_1, \dots, w_i as $fsck_{[i]}(d)$. By definition, $fsck(d) = fsck_{[n]}(d)$.
2. Let d_i be the disk image obtained by applying the writes w_1, \dots, w_i to disk image d_0 . This is the disk image returned by $fsck_{[i]}(d_0)$. Next, rerun `fsck` on d_i to verify that it produces the same file system as running it on d_0 (i.e., $fsck(d_i) = fsck(d_0)$). Computing $fsck(d_i) \equiv fsck(fsck_{[i]}(d_0))$ simulates the effect of a crash during the recovery where `fsck` performed i writes and then was restarted.

To illustrate, if invoking $fsck(d_0)$ writes two sectors 1 and then 4 with values v_1 , and v_2 respectively, the algorithm will first apply the write $\langle 1, v_1 \rangle$ to d_0 to obtain d_1 , crash, check, and then apply write $\langle 4, v_2 \rangle$ to d_1 to obtain d_2 , crash and check.

This approach requires three refinements before it is reasonable. The first is for speed, the second to catch more errors, and the third to reason about them. We describe all three below.

6.1 Speed From Determinism

The naive algorithm checks many more cases than it needs to. We can dramatically reduce this number by exploiting two facts. First, for all practical purposes we can regard `fsck` as a deterministic procedure (§5.4). Determinism implies a useful property: if two invocations of a deterministic function read the same input values, then they must compute the same result. Thus, if a given write by `fsck` does *not change* any value it previously read then there is no need to crash and rerun it — it will always get back to the current state. Second, `fsck` rarely writes data it reads. As a result, most writes do not require that we crash and recover: they will not intersect the set of sectors `fsck` reads and thus, by determinism, cannot influence the disk it would produce.

We state this independence more precisely as follows. Let $RS_i = \{r_1, \dots, r_k\}$ denote the (unordered) set of all sectors read by $fsck_{[i]}(d)$. As above, let d_i denote the disk produced by applying the writes (w_1, \dots, w_i) in order to the initial disk d_0 . We claim that if the sector s_k written by w_k is not in the read set RS_i , then running `fsck` to completion on disk d_i produces the same result as running it on d_{i-1} . I.e., $s_i \notin RS_i$ implies $fsck(d_i) = fsck(d_{i-1})$ (recall that $RS_{i-1} \subseteq RS_i$). Tautologically, a deterministic function can only change what it computes if the values it reads are different. Thus, $s_i \notin RS_i$ implies that $fsck(d_i)$ and $fsck(d_{i-1})$ read identical values for the first $i - 1$ steps, forcing them to both compute the same results so far. Then, at step i , both will perform write w_i , making their disks identical.

There are two special cases that our checker exploits to skip running `fsck`:

1. Suppose w_i does write a sector in RS_i , but the value it writes is the same as what is currently on disk (i.e., $d_i = d_{i-1}$). Clearly if $d_{i-1} = d_i$ then $fsck(d_i) = fsck(d_{i-1})$. Surprisingly, recovery programs empirically do many such redundant writes.
2. If (1) the sector s_i written by w_i is dominated by an earlier write w_j and (2) there is no read that precedes w_j , then w_i cannot have any affect since s_i will always be overwritten with v_j when `fsck` is restarted.

6.2 Checking All Write Orderings

As described the algorithm can miss many errors. Sector writes can complete in any order unless (1) they are ex-

licitly done synchronously (e.g., using the `O_DIRECT` option on Unix) or (2) they are blocked by a “sync barrier,” such as the `sync` system call on Unix which (is believed to) only return when all dirty blocks have been written to disk. Thus, generating all disk images possible after crashing `fsck` at any point requires partitioning the writes into “sync groups” and checking that `fsck` would work correctly if it was rerun after performing any subset of the writes in a sync group (i.e., the power set of the writes contained in the sync group). For example, if we write sectors 1 and 2, call `sync`, then write sector 4, we will have two sync groups $S_0 = \{1, 2\}$ and $S_1 = \{4\}$. The first, S_0 , generates three different write schedules: $\{(1), (2), (1, 2)\}$. A write schedule is the sectors that were written before `fsck` crashed (note that the schedule $(2, 1)$ is equivalent to $(1, 2)$ since we rerun `fsck` after both writes complete). Given a sync group S_i our checker does one of two things.

1. If the size of S_i is less than or equal to a user-defined threshold, t , the checker will exhaustively verify all different interleavings.
 2. If the size is larger than t the checker will do a series of trials, where it picks a random subset of random size within S_i . These trials can be deterministically replayed later because we seed the pseudo-random number generator with a hash of the sectors involved.
- We typically set $t = 5$ and the number of random trials to 7. Without this reordering we found no bugs in `fsck` recovery; with it we have found them in all three checked file systems.

6.3 Finding the Right Perspective

Unfortunately, while recovery errors are important, reasoning about them is extremely difficult. For most recovery errors the information the model checker provides is of the form “you wrote block 17 and block 38 and now your disk image has no files below ‘/’.” Figuring out (1) semantically what was being done when this error occurred, (2) what the blocks are for, and (3) why writing these values caused the problem can easily take an entire day. Further, this process has to be replicated by the implementor who must fix it. Thus, we want to find the simplest possible error case. The checker has five modes, described below and roughly ordered in increasing degrees of freedom and hence difficulty in diagnosing errors. (Limiting degrees of freedom also means they are ordered by increasing cost.) At first blush, five modes might seem excessive. In reality they are a somewhat desperate attempt to find a perspective that makes reasoning about an error tractable. If we could think of additional useful views we would add them.

Synchronous, atomic, logical writes. The first, simplest view is to group all sector writes into “logical writes” and do these synchronously (i.e., in the order

that they occur in program execution). Here, logical writes means we group all blocks written by the same system call invocation as one group. I.e., if there are two calls to the `write` system call, the first writing sectors $l_0 = (1, 2, 3)$ and the second writing sectors $l_1 = (7, 8)$ we have two logical operations, l_0 and l_1 . We apply all the writes in l_0 , crash, check, apply the writes in l_1 , crash, and check.

This is the strongest logical view one can have of disk: all operations complete in the order they were issued and all the writes in a single logical operation occur atomically. It is relatively easy to reason about these errors since it just means that `fsck` was not reentrant.

Synchronous, non-atomic, left-to-right logical writes. Here we still treat logical writes as synchronous, but write their contained sectors non-atomically, left-to-right. I.e., write the first sector in a logical group, crash, check, then write the next, etc. These errors are also relatively easy to reason about and tend to be localized to a single invocation of a system call where a data structure that was assumed to be internally consistent straddled a sector boundary.

Reordered, atomic logical writes. This mode reorders all logical writes within the same sync group, checking each permutation. These errors can often be fixed by inserting a single `sync` call.

Synchronous, non-atomic logical writes. This mode writes the sectors within a logical operation in any order, crashing after each possible schedule. These errors are modular, but can have interesting effects.

Reordered sector writes. This view is the hardest to reason about, but the sternest test of file system recovery: reorder all sectors within a sync group arbitrarily. We do not like these errors, and if we hit them will make every attempt to find them with one of the prior modes.

7 Results

Table 2 summarizes the errors we found, broken down by file systems and categories. All errors were reported to the respective developers. We found 32 serious bugs in total; 21 have been fixed and 9 of the remaining 11 confirmed. The latter were complex enough that no patch has been submitted. There were 10 errors where supposedly stable, committed data and metadata (typically entire directories) could be lost. JFS has higher error counts in part due to the immediate responses from the JFS developers, which enabled us to patch the errors and continue checking JFS. We are currently checking the XFS file system and the preliminary results are promising.¹

We discuss the bug categories in more detail below, highlighting the more interesting errors found.

¹The errors reported in this paper can be found at page <http://keeda.stanford.edu/~junfeng/osdi-fisc-bugs.html>, titled “OSDI FiSC Bugs.”

Error type	VFS	ext2	ext3	JFS	ReiserFS	total
Lost stable data	n/a	n/a	1	8	1	10
False clean	n/a	n/a	1	1		2
Security holes		2	2 (minor)	1		5
Kernel crashes	1			10	1	12
Other (serious)	1		1	1		3
Total	2	2	5	21	2	32

Table 2: We found 32 errors, 10 of which could cause permanent data loss. There are 3 intended errors, where programmers decided to sacrifice consistency for availability. They are not shown in this table.

7.1 Unrecoverable Data Loss

The most serious errors we found caused the irrevocable loss of committed, stable data. There were 10 such errors where an execution sequence would lead to the complete loss of metadata (and its associated data) that was committed to the on-disk journal. In several cases, all or large parts of long-lived directories, including the root directory “/”, were obliterated. Data loss had two main causes: (1) invalid write ordering of the journal and data during recovery and (2) buggy implementations of transaction abort and `fsck`.

Invalid recovery write ordering. There were three bugs of this type. During normal operation of a journaling file system the journal must be flushed to disk before the data it describes. The file systems we check seem to get this right. However, they all get the inverse of this ordering constraint wrong: during recovery, when the journal is being replayed, all data modified by this roll forward must be flushed to disk before the journal is persistently cleared. Otherwise, if a crash occurs, the file system will be corrupt or missing data, but the journal will be empty and hence unable to repair the file system.

Figure 4 gives a representative error from the ext3 `fsck` program. The chain of mishaps is as follows:

1. `recover_ext3_journal` rolls the journal forward by calling `journal_recover`.
2. `journal_recover` replays the journal, writing to the file system using cached writes. It then calls `fsync_no_super` to flush all the modified data back to disk. However, this macro has been defined to do nothing due to an error made moving the recovery code out of the kernel and into a separate `fsck` process.
3. Control returns to `recover_ext3_journal` which then calls `e2fsck_journal_release` which writes the now cleared journal to disk. Unfortunately, the lack of sync barriers allows this write to reach disk before the modified data. As a result, a crash that occurs after this point can obliterate parts of the file system, but the journal will be empty, causing data loss.

When this bug was reported, the developers immedi-

```
// e2fsprogs-1.34/e2fsck/jfs_user.h
// Error: empty macro, does not sync data!
#define fsync_no_super(dev) do {} while(0)

// e2fsprogs-1.34/e2fsck/journal.c
static errcode_t recover_ext3_journal(e2fsck_t ctx) {
    journal_t *journal;
    int retval;

    journal_init_revoke_caches();
    retval = e2fsck_get_journal(ctx, &journal);
    /* ... */
    retval = -journal_recover(journal);
    /* ... */

    // Flushes empty journal.
    e2fsck_journal_release(ctx, journal, 1, 0);
    return retval;
}

// e2fsprogs-1.34/e2fsck/recovery.c
int journal_recover(journal_t *journal) {
    // process journal records using cached writes.
    err = do_one_pass(journal, &info, PASS_SCAN);
    if (!err)
        err = do_one_pass(journal, &info, PASS_REVOKE);
    if (!err)
        // writes persistent data recorded in
        // journal using cached write calls.
        err = do_one_pass(journal, &info, PASS_REPLAY);

    /* ... */

    // Write all modified data back before clearing journal.
    fsync_no_super(journal->j_fs_dev);
    return err;
}
```

Figure 4: Journal write ordering bug in ext3 `fsck`.

ately released a patch. ReiserFS and JFS both had similar bugs (both now fixed), but in these systems the code lacked any attempt to order the journal clear with the writes of journal data.

Buggy transaction abort and `fsck`. There were five bugs of this type, all in JFS. Their causes were threefold.

First, JFS immediately applies all journaled operations to its in-memory metadata pages. Unfortunately, doing so makes it hard to roll back aborted transactions since

their modifications may be interleaved with the writes of many other ongoing or committed transactions. As a result, when JFS aborts a transaction, it relies on custom code to carefully extricate the side-effects of the aborted transactions from non-aborted ones. If the writer of this code forgets to reverse a modification, it can be flushed to disk, interlacing many directories with invalid entries from aborted transactions.

Second, JFS's `fsck` makes no attempts to recover any valid entries in such directories. Instead its recovery policy is that if a directory contains a single invalid entry it will remove all the entries of the directory and attempt to reconnect subdirectories and files into "lost+found." This opens a huge vulnerability: any file system mistake that results in persistently writing an invalid entry to disk will cause `fsck` to deliberately destroy the violated directory.

Third, JFS `fsck` has an incorrect optimization that allows the loss of committed subdirectories and files. JFS dynamically allocates and places inodes for better performance, tracking their location using an "inode map." For speed, incremental modifications to this map are written to the on-disk journal rather than flushing the map to disk on every inode allocation or deletion. During reconstruction, the `fsck` code can cause the loss of inodes because while it correctly applies these incremental modifications to its copy of the inode map, it deliberately does not overwrite the out-of-date, on-disk inode map with its (correct) reconstructed copy.

Figure 5 shows this bug, which has been in the JFS code since the initial version of JFS `fsck` over three years ago. The implementors incorrectly believed that if the file system was marked dirty, flushing the inode map was unnecessary because it would be rebuilt later. While the fix is trivial (always flushing the map), this bug was hard to find without a model checker; the JFS developers believe they have been chasing manifestations of it for a while [23]. After we submitted the bug report with all the file system events (operations and sector writes) and choices made by the model checker, a JFS developer was able to create a patch in a couple of days. This was a good example of the fact that model checking improves on testing by being more systematic, repeatable, and better controlled.

Other data loss bugs. A JFS journal that spans three or more sectors has the following vulnerability. JFS stores a sequence number in both the first and last sector of its journal but not in the middle sectors. After a crash, JFS `fsck` checks that these sequence numbers match and, if so, replays the journal. Without additional checking, inopportune sector reorderings can obviously lead to a corrupt journal, which will badly mutilate the file system when replayed.

Both JFS and `ext3` had a bug where a crashed file

```
// jfsutils-1.1.5/libfs/logredo.c
/* [Original, incorrect comment]
 * don't update the maps if the aggregate is
 * FM_DIRTY since fsck will rebuild maps anyway */
if (!vopen[k].is_fsdirty) { // check dirtiness
    // update on-disk map
    if ((rc = updateMaps(k)) != 0) {
        fsck_send_msg(lrdo_ERRORCANTUPDMAPS);
        goto error_out;
    }
}
```

Figure 5: Incorrect JFS `fsck` optimization which causes unrecoverable loss of inodes and their associated data.

system's superblock could be falsely marked as "clean." Thus, their `fsck` program would not repair the system, potentially leading to data loss or a system crash.

The last data loss bug happened when JFS incorrectly stored a negative error code as an inode number in a directory entry; this invalid entry would cause any later `fsck` invocation to remove the directory.

7.2 Security Holes

While we did not target security, FiSC found five security holes, three of which appear readily exploitable.

The easiest exploit we found was a storage leak in the JFS routine `jfs_link` used to create hard links. It calls the routine `get_UCSname`, which allocates up to 255 bytes of memory. `jfs_link` must (but does not) free this storage before returning. This leak occurs each time `jfs_link` is called, allowing a user to trivially do a denial of service attack by repeatedly creating hard links. Even ignoring malice, leaking storage on each hard link creation is generally bad.

The two other seemingly exploitable errors both occurred in `ext2` and were both caused by lookup routines that did not distinguish between lookups that failed because (1) no entry existed or (2) memory allocation failed. The first bug allows an attacker to create files or directories with the same name as a preexisting file or directory, hijacking all reads and writes intended for the original file. The second allows a user to delete non-empty directories to which they do not have write access.

In the first case, before creating a new directory entry, `ext2` will call the routine `ext2_find_entry` to see if the entry already exists. If `ext2_find_entry` returns NULL, the directory entry is created, otherwise it returns an error code. Unfortunately, in low memory conditions `ext2_find_entry` can return NULL even if the directory entry exists. As shown in Figure 6, the routine iterates over all pages in a directory. If page allocation fails (`ext2_get_page` returns NULL) it will skip this directory worth of entries and go to the next. Under low memory, `ext2_get_page` will always fail, no entry will be checked, and `ext2_find_entry` will always return NULL. This allows a user with write ac-

```

// linux-2.4.19/ext2/dir.c
struct ext2_dir_entry_2 * ext2_find_entry (struct inode * dir,
                                           struct dentry *dentry, struct page ** res_page)
{
    unsigned long start, n;
    unsigned long npages = dir_pages(dir);
    struct page *page = NULL;
    /* ... */
    // Iterate through all pages of the directory
    do {
        page = ext2_get_page(dir, n);
        if (!IS_ERR(page)) {
            // Code to check entry existence
            // Return the corresponding entry once found.
            /* ... */
        }
        // BUG: Error return from ext2_get_page ignored
    } while (...);
    return NULL;
    /* ... */
}

```

Figure 6: Ext2 security hole in `ext2_find_entry`.

cess to the directory to effectively create files and subdirectories with the same name as an existing file, hijacking all reads and writes intended for the original file.

The second error was similar, `ext2_rmdir` calls the routine `ext2_empty_dir` to ensure that the target directory is empty. Unfortunately the return value of `ext2_empty_dir` is the same if either the directory has no entries or if memory allocation fails, allowing an attacker to delete non-empty directories when they should not have permission to do so.

The remaining two errors occurred in `ext3` and were identical to the `ext2` bugs except that they were caused by disk read errors rather than low-memory conditions.

7.3 Other Bugs

Kernel crashes. There were 12 bugs which caused the kernel to crash because of a null pointer dereference. Most of these errors were due to improperly handled allocation failures. There was one error in the VFS layer, one error in ReiserFS, and 10 in JFS. The most interesting error was in JFS where `fsck` failed to correctly repair a file system, but marked it as clean. A subsequent traversal of the file system would panic the kernel.

Incorrect code. There were two cases where code just did the wrong thing. For example, `sys_create` creates a file on disk, but returns an error if a subsequent allocation fails. The application will think the file has not been created when it has. This error was interesting since it was in very heavily tested code in the VFS layer shared by all file systems.

Leaks. In addition to the leak mentioned above, the JFS routine `jfs_unmount` leaks memory on every unmount of a file system.

8 Experience

This section describes some of our experiences with FiSC: its use during development, sources of false positives and false negatives, and design lessons learned.

8.1 FiSC-Assisted Development

We checked preexisting file systems, and so could not comprehensively study how well model checking helps the development process. However, the responsiveness of the JFS developers allowed us to do a micro-case study of FiSC-assisted software development by following the evolution of a series of mistaken fixes:

1. We found and reported two kernel panics in the JFS transaction abort function `txAbortCommit` when called by the transaction commit function `txCommit` if memory allocation failed.
2. A few days later, the JFS developers sent a patch that removed `txAbortCommit` entirely and made `txCommit` call `txAbort` instead.
3. We applied the patch and replayed the original model checking sequence and verified it fixed the two panics. However, when we ran full model checking, within seconds we got segmentation faults in the VFS code. Examination revealed that the newly created inode was inserted into the VFS directory entry cache before the transaction was committed. A failed commit freed the inode and left a dangling pointer in the VFS directory entry cache. We sent this report back to the JFS developers.
4. As before: a few days later, they replied with a second patch, we applied it, it again fixed the specific error that occurred. We ran FiSC on the patched code and found a new error, where `fsck` would complain that a parent directory contained an invalid entry and would remove the parent directory entirely. This was quite a bit worse than the original error.
5. This bug is still outstanding.

While there are many caveats that one must keep in mind, model checking has some nice properties. First, it makes it trivial to verify that the original error is fixed. Second, it allows more comprehensive testing of patches than appears to be done in commercial software houses. Third, it finds the corner-case implications of seemingly local changes in seconds and demonstrates that they violate important consistency invariants.

8.2 False Positives

The false positives we found fell into two groups. Most were bugs in the model checking harness or in our understanding of the underlying file system and not in the checked code itself. The latter would hopefully be a minor problem for file system implementors using our system (though it would be replaced by problems arising from their imperfect understanding of the underlying model checker). We have had to iteratively correct a

series of slight misunderstandings about the internals of each of the file systems we have checked.

The other group of false positives stemmed from implementors intentionally ignoring or violating the properties we check. For example, ReiserFS causes a kernel panic when disk read fails in certain circumstances. Fortunately, such false positives are easily handled by disabling the check.

8.3 False Negatives

In the absence of proving total correctness, one can always check more things. We are far from verification. We briefly describe what we believe are the largest sources of missed errors.

Exploring thresholds. We do a poor job of triggering system behavior that only occurs after crossing a threshold value. The most glaring example: because we only test a small number of files and directories (≤ 15) we miss bugs that happen when directories undergo reorganization or change representations only after they contain a “sufficient” number of entries. Real examples include the re-balancing of directory tree structures in JFS or using a hashed directory structure in ext3. With that said, FiSC does check a mixture of large and small files (to get different inode representations) and file names or directories that span sector boundaries (for crash recovery).

Multi-threading support. The model checker is single-threaded both above and below the system call interface. Above, because only a single user process does file system operations. Below, because each state transition runs atomically to completion. This means many interfering state modifications never occur in the checked system. In particular, in terms of high-level errors, file system operations never interleave and, consequently, neither do partially completed transactions (either in memory or on disk). We expect both to be a fruitful source of bugs.

White-box model checking. FiSC can only flag errors that it sees. Because it does not instrument code it can miss low-level errors, such as memory corruption, use of freed memory, or a race condition unless they cause a crash or invariant violation. Fortunately, because we model-check implementation code we can simultaneously run dynamic tools on it.

Unchecked guarantees. File systems provide guarantees that are not handled by our current framework. These include versioning, undelete operations, disk quotas, access control list support, and journaling of data or, in fact, any reasonable guarantees of data block contents across crashes. The latter is the one we would like to fix the most. Unfortunately, because of the lack of agreed-upon guarantees for non-`sync`d data across crashes we currently only check metadata consistency across crashes — data blocks that do not precede a “`sync`” point can be

corrupted and lost without complaint.

File systems are directed acyclic graphs, and often trees. Presumably events (file system operations, failures, bad blocks) should have topological independence — events on one subgraph should not affect any other disjoint subgraph. Events should also have temporal independence in that creating new files and directories should not harm old files and directories.

One way to broaden the invariants we check would be to infer FS-specific knowledge using the techniques in [29].

Missed states. While our state hashing (§5.1) can potentially discard too much detail, we do not currently discard enough of the right details, possibly missing real errors. Using FS-specific knowledge opens up a host of additional state optimizations. One profitable example would be if we knew which interleavings of buffer cache blocks and `fsck` written blocks are independent (e.g., those for different files), which would dramatically reduce the number of permutations needed for checking the effects of a crash.

We have not aggressively verified statement coverage, so all file systems almost certainly contain many unexercised statements.

8.4 Design Lessons

One hard lesson we learned was a sort of “Heisenberg” principle of checking: make sure the inspection done by your checking code does not perturb the state of the checked system. Violating this principle leads to mysterious bugs. A brief history of the code for traversing a mounted file system and building a model drives this point home.

Initially, we extracted the VolatileFS by using a single block device that the test driver first mutated and then traversed to create a model of the volatile file system after the mutation. This design deadlocked when a file system operation did a multi-sector write and the traversal code tried to read the file system after only one of the sectors was written. The file system code responsible for the write holds a lock on the file being written, a lock that the traversal code wants to acquire but cannot. We removed this specific deadlock by copying the disk after a test driver operation and then traversing this copy, essentially creating two file systems. This hack worked until we started exploring larger file system topologies, at which point we would deadlock again because the creation of the second file system copy would use all available kernel memory, preventing the traversal thread from being able to successfully complete. Our final hack to solve this problem was to create a reserve memory pool for the traversal thread.

In retrospect, the right solution is to run two kernels side by side: one dedicated to mutating the disk,

the other to inspecting the mutated disk. Such isolation would straightforwardly remove all perturbations to the checked system.

A similar lesson is that the system being checked should be instrumented instead of modified unless absolutely necessary. Code always contains hidden assumptions, easily violated by changing code. For example, the kernel we used had had its kernel memory allocators re-implemented in previous work [25] as part of doing leak checking. While this replacement worked fine in the original context of checking TCP, it caused the checked file systems to crash. It turned out they were deliberately mangling the address of the returned memory in ways that intimately depended on how the original allocator (`page_alloc`) worked. We promptly restored the original kernel allocators.

9 Related Work

In this section, we compare our approach to file system testing techniques, software model checking efforts and other generic bug finding approaches.

File system testing tools. There are many file system testing frameworks that use application interfaces to stress a “live” file system with an adversarial environment. While these frameworks are less comprehensive than model checking they require much less work than that required to jam an entire OS into a model checker. We view testing as complementary to model checking — there is no reason not to test a file system and then apply model checking (or vice versa). It is almost always the case that two different but effective tools will find different errors, irrespective of their theoretical strengths and weaknesses.

Software Model Checking. Model checkers have been previously used to find errors in both the design and the implementation of software systems [1, 3, 6, 18–20].

We compare our work with two model checkers that are the most similar to our approach, both of which execute the system implementation directly without resorting to an intermediate description.

Verisoft [18] is a software model checker that systematically explores the interleavings of a concurrent C program. Unlike the CMC model checker we use, Verisoft does not store states at checkpoints and thereby can potentially explore a state more than once. Verisoft relies heavily on partial order reduction techniques that identify (control and data) independent transitions to reduce the interleavings explored. Determining such independent transitions is extremely difficult in systems with tightly coupled threads sharing large amount of global data. As a result, Verisoft would not perform well for these systems, including the Linux file systems checked in this paper.

Java PathFinder [3] is very similar to CMC and sys-

tematically checks concurrent Java programs by checkpointing states. It relies on a specialized virtual machine that is tailored to automatically extract the current state of a Java program. The techniques described in this paper are applicable to Java Pathfinder as well.

Generic bug finding. There has been much recent work on bug finding, including both better type systems [9, 14, 16] and static analysis tools [1, 4, 7, 8, 11, 15]. Roughly speaking [12], because static analysis can examine all paths and only needs to compile code in order to check it, it is relatively better at finding errors in surface properties visible in the source (“`lock` is paired with `unlock`”). In contrast, model checking requires running code, which makes it much more strenuous to apply (days or weeks instead of hours) and only lets it check executed paths. However, because it executes code it can more effectively check the properties implied by code. (E.g., that the log contains valid records, that `fsck` will not delete directories it should not.) Based on our experiences using static analysis, the most serious errors in this paper would be difficult to get with that approach. But, as with testing, we view static analysis as complementary to model checking — it is lightweight enough that there is no reason not to apply it and then use model checking.

10 Conclusion

This paper has shown how model checking can find interesting errors in real file systems. We found 32 serious errors, 10 of which resulted in the loss of crucial metadata, including the file system root directory “/”. The majority of these bugs have resulted in immediate patches.

Given how heavily-tested the file systems we model-checked were and the severity of the errors found, it appears that model checking works well in the context of file systems. This was a relief — we have applied full system model-checking in other contexts less successfully [12]. The underlying reason for its effectiveness in this context seems to be because file systems must do so many complex things right. The single worst source of complexity is that they must be in a recoverable state in the face of crashes (e.g., power loss) at every single program point. We hope that model checking will show similar effectiveness in other domains that must reason about a vast array of failure cases, such as database recovery protocols and optimized consensus algorithms.

11 Acknowledgments

We thank Dave Kleikamp for answering our JFS related questions, diagnosing bugs and submitting patches, Andreas Dilger, Theodore Ts'o, Al Viro, Christopher Li, Andrew Morton, Stephen C. Tweedie for their help with ext2 and ext3, Oleg Drokin and Vitaly Fertman for ReiserFS. We especially thank Ted Kremenek for last-minute

comments and edits. We are also grateful to Andrew Myers (our shepherd), Ken Ashcraft, Brian Gaeke, Lea Kissner, Ilya Shpitser, Xiaowei Yang, Monica Lam and the anonymous reviewers for their careful reading and valuable feedback.

References

- [1] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001 Workshop on Model Checking of Software*, May 2001.
- [2] H.-J. Boehm. Simple garbage-collector-safety. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, 1996.
- [3] G. Brat, K. Havelund, S. Park, and W. Visser. Model checking programs. In *IEEE International Conference on Automated Software Engineering (ASE)*, 2000.
- [4] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, 2000.
- [5] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
- [6] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *ICSE 2000*, 2000.
- [7] SWAT: the Coverity software analysis toolset. <http://coverity.com>.
- [8] M. Das, S. Lerner, and M. Seigle. Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.
- [9] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, June 2001.
- [10] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*, pages 522–525. IEEE Computer Society, 1992.
- [11] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, Sept. 2000.
- [12] D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In *Invited paper: Fifth International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI04)*, pages 191–210, Jan. 2004.
- [13] The ext2/ext3 File system. <http://e2fsprogs.sf.net>.
- [14] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 219–232, 2000.
- [15] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245. ACM Press, 2002.
- [16] J. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, June 2002.
- [17] G. R. Ganger and Y. N. Patt. Soft updates: A solution to the metadata update problem in file systems. Technical report, University of Michigan, 1995.
- [18] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, 1997.
- [19] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.
- [20] G. J. Holzmann. From code to models. In *Proc. 2nd Int. Conf. on Applications of Concurrency to System Design*, pages 3–10, Newcastle upon Tyne, U.K., 2001.
- [21] The IBM Journaling File System for Linux. <http://www-124.ibm.com/jfs>.
- [22] M. K. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
- [23] D. Kleikamp. Private communication.
- [24] Linux Test Project. <http://ltp.sf.net>.
- [25] M. Musuvathi and D. R. Engler. Model checking large network protocol implementations. In *Proceedings of the First Symposium on Networked Systems Design and Implementation*, 2004.
- [26] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, 2002.
- [27] ReiserFS. <http://www.namesys.com>.
- [28] Sandberg, Goldberg, Kleiman, Walsh, and Lyon. Design and implementation of the Sun network file system, 1985.
- [29] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-smart disk systems. In *Second USENIX Conference on File and Storage Technologies*, 2003.
- [30] stress. <http://weather.ou.edu/~apw/projects/stress>.
- [31] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, 2002.

CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code

Zhenmin Li, Shan Lu, Suvda Myagmar and Yuanyuan Zhou

*Department of Computer Science
University of Illinois at Urbana-Champaign, Urbana, IL 61801*

ABSTRACT

Copy-pasted code is very common in large software because programmers prefer reusing code via copy-paste in order to reduce programming effort. Recent studies show that copy-paste is prone to introducing bugs and a significant portion of operating system bugs concentrate in copy-pasted code. Unfortunately, it is challenging to efficiently identify copy-pasted code in large software. Existing copy-paste detection tools are either not scalable to large software, or cannot handle small modifications in copy-pasted code. Furthermore, few tools are available to detect copy-paste related bugs.

In this paper we propose a tool, CP-Miner, that uses data mining techniques to efficiently identify copy-pasted code in large software including operating systems, and detects copy-paste related bugs. Specifically, it takes less than 20 minutes for CP-Miner to identify 190,000 copy-pasted segments in Linux and 150,000 in FreeBSD. Moreover, CP-Miner has detected 28 copy-paste related bugs in the latest version of Linux and 23 in FreeBSD. In addition, we analyze some interesting characteristics of copy-paste in Linux and FreeBSD, including the distribution of copy-pasted code across different length, granularity, modules, degrees of modification, and various software versions.

1 Introduction

1.1 Motivation

Copying and pasting code is a common practice in software development. In order to reduce programming effort and shorten programming time, programmers prefer reusing a piece of code via copy-paste rather than rewriting similar code from scratch. Recent studies [6, 13, 25] have shown that a large portion of code is duplicated in software. For example, Kapser and Godfrey [25], using a copy-paste detection tool called CCFinder [24], found that 12% of the Linux file system code (279K lines) was involved in code cloning activity. Baker [6] found that in the complete source of the X Window system (714K lines), 19% of the code was identified as duplicates.

Using abstractions such as functions and macros to remove this duplication might improve software maintenance; however, much duplication will likely remain, for

two possible reasons. First, some changes are usually necessary, and copy-paste is much easier and faster than abstraction. Another reason is that functions may impose higher overhead. However, the psychological reasons for large percentage of existing copy-pasted code are beyond the scope of this paper.

Copy-pasted code is prone to introducing errors. For example, Chou et al. [10] found that in a single source file under the Linux `drivers/i2o` directory, 34 out of 35 errors were caused by copy-paste. One of the errors was copied in 10 places and another in 24. They also showed that many operating system errors are not independent because programmers are ignorant of system restrictions in copy-pasted code. In our study, we have detected 28 copy-paste related bugs in the latest version of Linux and 23 in FreeBSD. Most of these bugs were previously unreported.

A major reason why copy-paste introduces bugs is that programmers forget to modify identifiers (variables, functions, types, etc.) consistently throughout the pasted code. This mistake will be detected by a compiler if the identifier is undefined or has the wrong type. However, these errors often slip through compile-time checks and become hidden bugs that are very hard to detect.

Figure 1 shows an example of a bug detected by CP-Miner in the latest version of Linux (2.6.6). We reported this bug to the Linux kernel community and it has been confirmed by kernel developers [1]. In this example, the loop in lines 111–118 was copied from lines 92–99. In the new copy-pasted segment (lines 111–118), the variable `prom_phys_total` is replaced with `prom_prom_taken` in most of the cases except the one in line 117 (shown in bold font). As a result, the pointer `prom_prom_taken[iter].theres_more` incorrectly points to the element of `prom_phys_total` instead of `prom_prom_taken`. This bug is a semantic error, and therefore it cannot be easily detected by memory-related bug detection tools including static checkers [9, 14, 17, 32] or dynamic tools such as Purify [19], Valgrind [36], and CCured [12]. Besides this bug, CP-Miner has also detected many other similar bugs caused by copy-paste in Linux, FreeBSD, PostgreSQL and Web Apache.

While one can imagine augmenting the software development tools and editors with copy-paste tracking, this support does not currently exist. Therefore, we are fo-

```

( linux-2.6.6/arch/sparc64/prom/memory.c )
68 void __init prom_meminit(void)
69 {
    *****
92  for(iter=0; iter<num_regs; iter++) {
93      prom_phys_total[iter].start_adr =
94      prom_reg_memlist[iter].phys_addr;
95      prom_phys_total[iter].num_bytes =
96      prom_reg_memlist[iter].reg_size;
97      prom_phys_total[iter].theres_more =
98      &prom_phys_total[iter+1];
99  }
    *****
111 for(iter=0; iter<num_regs; iter++) {
112     prom_prom_taken[iter].start_adr =
113     prom_reg_memlist[iter].phys_addr;
114     prom_prom_taken[iter].num_bytes =
115     prom_reg_memlist[iter].reg_size;
116     prom_prom_taken[iter].theres_more =
117     &prom_phys_total[iter+1]; // bug
118 }
    *****
143 }

```

Figure 1: An example of a copy-paste related error detected by CP-Miner. This bug appears in `linux-2.6.6/arch/sparc64/prom/memory.c`. A similar bug is also detected in file `/arch/sparc/prom/memory.c`.

cusing on detecting likely copied and pasted code in an existing code base. Not all code segments identified by previous detection tools and our tool are really the results of copy-paste (even though we prune many of the false copy-pasted segments as described in Section 3.1.4), but for simplicity we refer likely-copy-pasted segments as copy-pasted segments.

It is a challenging task to efficiently extract copy-pasted code in large software such as an operating system. Even though some previous studies [16, 20] have addressed the related problem of plagiarism detection, they are not suitable for detecting copy-pasted code. Those tools, such as the commonly used JPlag [33], were designed to measure the degree of similarity between a pair of programs in order to detect cheating. If these tools were to be used to detect copy-pasted code in a single program *without any modification*, they would need to compare all possible pairs of code fragments. For a program with n statements, a total of $O(n^3)$ pairwise comparisons¹ would need to be performed. This complexity is certainly impractical for software with millions of lines of code such as Linux and FreeBSD. Of course, it is possible to modify these tools to identify copy-pasted code in single software, but the modification is not trivial and straightforward.

So far, only a few tools have been proposed to identify copy-pasted code in a single program. Examples of such tools include Moss [4, 35], Dup [6], CCFinder [24] and

¹Considering comparison between the pair of code fragments with k statements, there are $(n - k + 1)$ different fragments. So there are $\binom{n-k+1}{2} = O(n^2)$ possible pair comparisons. Since k can be 1, 2, ..., $\frac{n}{2}$, the total number of pairwise comparisons is $O(n^3)$.

others [5, 7]. Most of these tools suffer from some or all of the following limitations:

(1) *Efficiency*: Most existing tools are not scalable to large software such as operating system code because they consume a large amount of memory and take a long time to analyze millions of lines of code.

(2) *Tolerance to modifications*: Most tools cannot deal with modifications in copy-pasted code. Some tools [13, 22] can only detect copy-pasted segments that are exactly identical. Moreover, most of the existing tools do not allow statement insertions or modifications in a copy-pasted segment. Such modifications are very common in standard practice. Our experiments with CP-Miner show that about one third of copy-pasted segments contain insertion or modification of 1-2 statements.

(3) *Bug detection*: The existing tools cannot detect copy-paste related bugs. They only aim at detecting copy-pasted code and do not consider bugs associated with copy-paste.

1.2 Our Contributions

In this paper we present CP-Miner, a tool that uses data mining techniques to *efficiently* identify copy-pasted code in large software including operating system code, and also detects copy-paste related bugs. It requires no modification or annotation to the source code of software being analyzed. Our paper makes three main contributions:

(1) **A scalable copy-paste detection tool for large software**: CP-Miner can efficiently find copy-pasted code in large software including operating system code. Our experimental results show that it takes less than 20 minutes for CP-Miner to detect 150,000–190,000 different copy-pasted segments that account for about 20–22% of the source code in Linux and FreeBSD (each with more than 3 million lines of code). Additionally, it takes less than one minute to detect copy-pasted segments in Apache web server and PostgreSQL, accounting for about 17–22% of total source code.

Compared to CCFinder [24], CP-Miner is able to find 17–52% more copy-pasted segments because CP-Miner can tolerate statement insertions and modifications.

(2) **Detection of bugs associated with copy-paste**: CP-Miner can detect copy-paste related bugs such as the one shown in Figure 1, most of which are hard to detect with existing static or dynamic bug detection tools. More specifically, CP-Miner has detected 28 potential bugs in the latest version of Linux, 23 in FreeBSD, 5 in Web Apache, and 2 in PostgreSQL. Most of these bugs had never been reported.

We have reported these bugs to the corresponding developers. So far five bugs have recently been confirmed and fixed by Linux developers, and one bug has been confirmed and fixed by Apache developers.

(3) **Statistical study of copy-pasted code distribution in operating system code**: Few previous studies have been

conducted on the characteristics of copy-paste in large software. Our work analyzed some interesting statistics of copy-pasted code in Linux and FreeBSD. Our results indicate that (1) copy-pasted segments are usually not too large, most with 5–16 statements; (2) although more than 50% of copy-pasted segments have only two copies, a few (6.3–6.7%) copy-pasted segments are copied more than 8 times; (3) there is a significant number (11.3–13.5%) of copy-pasted segments at function granularity (copy-paste of an entire function); (4) most (65–67%) copy-pasted segments require renaming at least one identifier, and 23–27% of copy-pasted segments have inserted, modified, or deleted one statement; (5) different OS modules have very different copy-paste coverage: *drivers*, *arch*, and *crypt* have higher percentage of copy-paste than other modules in Linux; (6) as the operating system code evolves, the amount of copy-paste also increases, but the coverage percentage of copy-pasted code remains relatively stable over the recent versions of Linux and FreeBSD.

2 Background

2.1 Detection of Copy-pasted Code

Since copy-pasted code segments are usually similar to the original ones, detection of copy-pasted code involves detecting code segments that are identical or similar.

Previous techniques for copy-paste detection can be roughly classified into three categories: (1) *string-based*, in which the program is divided into strings (typically lines), and these strings are compared against each other to find sequences of duplicated strings [6]; (2) *parse-tree-based*, in which pattern matching is performed on the parse-tree of the code to search for similar subtrees [7, 27]; (3) *token-based*, in which the program is divided into a stream of tokens and duplicate token sequences are identified [24, 33].

Our tool, CP-Miner, is token-based. This approach has advantages over the other two. First, a string-based approach does not exploit any lexical information, so it cannot deal with simple modifications such as identifier renaming. Second, using parse trees can introduce false positives because two segments with identical syntax trees are not necessarily copy-pasted. This is because copy-paste is code-based rather than syntax-based, i.e., it reuses a piece of code rather than an abstract syntax structure.

Most previous copy-paste detection tools do not sufficiently address the limitations described in Section 1. Most of them consume too much time or memory to be scalable to large software, or do not tolerate modifications made in copy-pasted code. In contrast, CP-Miner can address both challenges.

2.2 Frequent Subsequence Mining

CP-Miner is based on *frequent subsequence mining* (also called frequent sequence mining), an association analysis

technique that discovers frequent subsequences in a sequence database [2]. Frequent subsequence mining is an active research topic in data mining [38, 39]. It has broad applications, including mining motifs in DNA sequences, analysis of customer shopping behavior, etc.

A subsequence is considered *frequent* when it occurs in at least a specified number of sequences (called *min.support*) in the sequence database. A subsequence is not necessarily contiguous in an original sequence. We denote the number of occurrences of a subsequence as its *support*. A sequence that contains a given subsequence is called a *supporting sequence* of this subsequence.

For example, a sequence database D has five sequences: $D = \{abced, abecf, agbch, abijc, aklec\}$. The number of occurrences of subsequence abc is 4, and sequence $agbch$ is one of abc 's supporting sequences. If *min.support* is specified as 4, the frequent subsequences are $\{a: 5, b: 4, c: 5, ab: 4, ac: 5, bc: 4, abc: 4\}$, where the numbers are the supports of the subsequences.

CP-Miner uses a recently proposed frequent subsequence mining algorithm called *CloSpan* (Closed Sequential Pattern Mining)[38], which outperforms most previous algorithms. Instead of mining the complete set of frequent subsequences, CloSpan mines only the *closed subsequences*. A closed subsequence is the subsequence whose support is different from that of its super-sequences. CloSpan mainly consists of two stages: (1) using a depth-first search procedure to generate a candidate set of frequent subsequences that includes all the closed frequent subsequences; and (2) pruning the non-closed subsequences from the candidate set. The computational complexity of CloSpan is $O(n^2)$ if the maximum length of frequent sequences is constrained by a constant.

Mining efficiency in CloSpan is improved by two main ideas. The first is based on an observation that if a sequence is frequent, all of its subsequences are frequent. For example, if abc is frequent, all of its subsequences $\{a, b, c, ab, ac, bc\}$ are also frequent. CloSpan recursively produces a longer frequent subsequence by concatenating every frequent item to a shorter frequent subsequence that has already been obtained in the previous iterations.

Let us consider an example. Let L_n denote the set of frequent subsequences with length n . In order to get L_n , we can join the sets L_{n-1} and L_1 . For example, suppose we have already computed L_1 and L_2 as shown below. In order to compute L_3 , we can first compute L'_3 by concatenating a subsequence from L_2 and an item from L_1 :

$$\begin{aligned} L_1 &= \{a, b, c\}; \\ L_2 &= \{ab, ac, bc\}; \\ L'_3 &= L_2 \times L_1 = \{abc, abb, abc, aca, acb, acc, bca, bcb, bcc\} \end{aligned}$$

For greater efficiency, CloSpan does not join the sequences in set L_2 with all the items in L_1 . Instead, each sequence in L_2 is concatenated with only the frequent items in its *suffix database*. A suffix database of a subsequence s is the database of all the maximum suffixes of the sequences that contain s . In our example,

for the frequent sequence ab in L_2 , its suffix database is $D_{ab} = \{ced, cef, ch, ijc\}$, and only c is a frequent item, so ab is only concatenated with c and we get a longer sequence abc that belongs to L'_3 .

The second idea for improving mining performance is to efficiently evaluate whether a concatenated subsequence is frequent. Rather than searching the whole database, CloSpan only checks certain suffixes. In our example, for each sequence s in L'_3 , CloSpan checks whether it is frequent or not by searching the suffix database D_s . If the number of its occurrences is greater than min_sup , s is added into L_3 , which is the set of frequent subsequences of length 3. CloSpan continues computing L_4 from L_3 , L_5 from L_4 , and so on until no more subsequences can be added into the set of frequent subsequences.

Due to space limitation, a detailed discussion of the CloSpan algorithm can be found in [29, 38].

3 CP-Miner

CP-Miner has two major functionalities: detecting copy-pasted code segments, and finding copy-paste related bugs. It requires no modification to the source code of software being analyzed. The following two subsections describe the design for each functionality.

3.1 Identifying Copy-pasted Code

To detect copy-pasted code, CP-Miner first converts the problem into a frequent subsequence mining problem. It then uses an enhanced algorithm of CloSpan to find *basic* copy-pasted segments. Finally, it prunes false positives and composes larger copy-pasted segments. For convenience of description, we refer to a group of code segments that are similar to each other as a *copy-paste group*.

CP-Miner can detect copy-pasted segments *efficiently* because it uses frequent subsequence mining techniques that can avoid many unnecessary or redundant comparisons. To map our problem to a frequent subsequence mining problem, CP-Miner first maps a statement to a number, with similar statements being mapped to the same number. Then, a basic block (i.e., a straight-line piece of code without any jumps or jump targets in the middle) becomes a sequence of numbers. As a result, a program is mapped into a database of many sequences. By mining the database using CloSpan, we can find frequent subsequences that occur at least twice in the sequence database. These frequent subsequences are exactly copy-pasted segments in the original program. By applying some pruning techniques such as identifier mapping, we can find basic copy-pasted segments, which can then be combined with neighboring ones to compose larger copy-pasted segments.

CP-Miner is capable of handling modifications in copy-pasted segments for two reasons. First, similar statements are mapped into the same value. This is achieved by mapping all identifiers (variables, functions and types) of the same type into the same value, regardless of their actual

names. This relaxation tolerates identifier renaming in copy-pasted segments. Even though false positives may be introduced during this process, they are addressed later through various pruning techniques such as identifier mapping (described in Section 3.1.4). Second, we have enhanced the basic frequent subsequence mining algorithm, CloSpan, to support gap constraints in frequent subsequences. This enhancement allows CP-Miner to tolerate 1–2 statement insertions, deletions, or modifications in copy-pasted code. Insertions and deletions are symmetric because a statement deletion in one copy can also be seen as an insertion in the other copy. Modification is a special case of insertion. Basically, the modified statement can be treated as if both segments have a statement inserted.

The main steps of the process to identify copy-pasted segments include:

- (1) *Parsing source code*: Parse the given source code and build a sequence database (a collection of sequences). In addition, information regarding basic blocks and block nesting levels are also passed to the mining algorithm.
- (2) *Mining for basic copy-pasted segments*: The enhanced frequent subsequence mining algorithm is applied to the sequence database to find basic copy-pasted segments.
- (3) *Pruning false positives*: Various techniques including identifier mapping are used to prune false positives.
- (4) *Composing larger copy-pasted segments*: Larger copy-pasted segments are identified by combining consecutive smaller ones. The combined copy-pasted segments are fed back to step (3) to prune false positives. This is necessary because the combined one may not be copy-pasted, even though each smaller one is.

Like other copy-paste detection tools, CP-Miner can only detect copy-pasted segments, but cannot tell which segment is original and which is copy-pasted from the original. Fortunately, this limitation is not a big problem because in most cases it is enough for programmers to know what segments are similar to each other. Moreover, our bug detection method described in Section 3.2 does not rely on such differentiation. Additionally, if programmers really need the differentiation, navigating through RCS versions could help figuring out which segment is the original copy.

3.1.1 Parsing Source Code

The main purpose of parsing source code is to build a sequence database (a collection of sequences) in order to convert the copy-paste detection problem to a frequent subsequence mining problem. Comments are not considered normal statements in CP-Miner, and are thereby filtered by our parser. The current prototype of the CP-Miner parser only works for programs written in C or C++, but it is easy to modify it for other programming languages.

A statement is mapped to a number by first tokenizing its components such as variables, operators, constants, functions, keywords, etc. To tolerate identifier renaming in copy-pasted segments, identifiers of the same type are

mapped into the same token. Constants are handled in the same way as identifiers: constants of the same type are mapped into the same token. However, operators and keywords are handled differently, with each one mapped to a unique token. After all the components of a statement are tokenized, a hash value digest is computed using the “hashpjw” [3] hash function, chosen for its low collision rate. Figure 2 shows the hash value for each statement in the example shown in Figure 1 of Section 1. As shown in this figure, the statement in lines 93–94 and the statement in lines 112–113 have the same hash values.

After each statement is mapped, the program becomes a long number sequence. Unfortunately, the frequent subsequence mining algorithms need a collection of sequences (a sequence database) as described in 2.2, so we need a way to cut this long sequence into many short ones. One simple method is to use a fixed cutting window size (e.g., every 20 statements) to break the long sequence into many short ones. This method has two disadvantages. First, some frequent subsequences across two or more windows may be lost. Second, it is not easy to decide the window size: if it is too long, the mining algorithm would be very slow; if too short, too much information may be lost on the boundary of two consecutive windows.

Instead, CP-Miner uses a more elegant method to perform the cutting. It takes advantage of some simple syntax information and uses a basic programming block as the unit to break the long sequence into short ones. The idea for this cutting method is that a copy-pasted segment is usually either a part of a basic block or consists of multiple basic blocks. In addition, basic blocks are usually not too long to cause performance problems in CloSpan. By using a basic block as the cutting unit, CP-Miner can first find basic copy-pasted segments and then compose larger ones from smaller ones. Since different basic blocks have a different number of statements, their corresponding sequences also have different length. But this is not a problem for CloSpan because it can deal with sequences of different sizes. The example shown in Figures 1 and 2 is converted into the following collection of sequences:

```
(35487793)
```

```
*****
```

```
(67641265)
```

```
(133872016, 133872016, 82589171)
```

```
*****
```

```
(67641265)
```

```
(133872016, 133872016, 82589171)
```

```
*****
```

Besides a collection of sequences, the parser also passes to the mining algorithm the source code information of each sequence. Such information includes (1) the nesting level of each basic block, which is later used to guide the composition of larger copy-pasted segments from smaller ones; (2) the file name and line number, which is used to locate the copy-pasted code corresponding to a frequent subsequence identified by the mining algorithm.

STATEMENT	HASH
68 void __init prom_meminit(void)	35487793
69 {	
*****	*****
92 for(iter=0; iter<num_regs; iter++) {	67641265
93 prom_phys_total[iter].start_adr =	133872016
94 prom_reg_memlist[iter].phys_addr;	
95 prom_phys_total[iter].num_bytes =	133872016
96 prom_reg_memlist[iter].reg_size;	
97 prom_phys_total[iter].theres_more =	82589171
98 &prom_phys_total[iter+1];	
99 }	
*****	*****
111 for(iter=0; iter<num_regs; iter++) {	67641265
112 prom_prom_taken[iter].start_adr =	133872016
113 prom_reg_memlist[iter].phys_addr;	
114 prom_prom_taken[iter].num_bytes =	133872016
115 prom_reg_memlist[iter].reg_size;	
116 prom_prom_taken[iter].theres_more =	82589171
117 &prom_phys_total [iter+1];	
118 }	
*****	*****
143 }	

Figure 2: An example of hashing statements

3.1.2 Mining for Basic Copy-pasted Segments

After CP-Miner parses the source code of a given program, it generates a sequence database with each sequence representing a basic block. At the next step, it applies the frequent subsequence mining algorithm, CloSpan, on this database to find frequent subsequences with support value of at least 2, which corresponds to code segments that have appeared in the program at least twice. In the example shown in Figure 2, CP-Miner would find (133872016, 133872016, 82589171) as a frequent subsequence because it occurs twice in the sequence database. Therefore, the corresponding code segments in line 111–118 and line 92–99 are basic copy-pasted segments.

Unfortunately, the mining process is not as straightforward as expected. The main reason is that the original CloSpan algorithm was not designed exactly for our purpose, and nor were other frequent subsequence mining algorithms. Most existing algorithms including CloSpan have the following two limitations that we had to enhance CloSpan to make it applicable for copy-paste detection:

(1) Adding gap constraints in frequent subsequences: In most existing frequent subsequence mining algorithms, frequent subsequences are not necessarily contiguous in their supporting sequences. For example, sequence *abdec* provides 1 support for subsequence *abc*, even though *abc* does not appear contiguously in *abdec*. It is possible to have a large gap in the occurrence of a frequent subsequence in one of its supporting sequences. Hence, its corresponding code segment would have several statements inserted. Such segment is unlikely to be copy-pasted.

To address this problem, we modified CloSpan to add a gap constraint in frequent subsequences. CP-Miner only mines for frequent subsequences with a maximum gap not larger than a given threshold called *max_gap*. If the max-

imum gap of a subsequence in a sequence is larger than *max_gap*, this sequence is not “supporting” this subsequence. For example, for the sequence database $D = \{abced, abecf, agbch, abijc, aklc\}$, the support of subsequence *abc* is 1 if *max_gap* equals 0, and the support is 3 if *max_gap* equals 1.

The gap constraint with *max_gap* = 0 means that no statement insertion or deletions are allowed in copy-paste, whereas the gap constraint with *max_gap* = 1 or *max_gap* = 2 means that 1 or 2 statement insertions/deletions are tolerated in copy-paste.

(2) Matching frequent subsequences to copy-pasted segments: The original CloSpan algorithm outputs only frequent subsequences and their corresponding support values, but not their corresponding supporting sequences. To find copy-pasted code, we need to find the supporting sequences for each frequent subsequence.

We enhance CloSpan to address this problem. When CP-Miner generates a frequent subsequence, it maintains a list of IDs of its supporting sequences. In the above example, CP-Miner outputs two frequent subsequences: (67641265) and (133872016, 133872016, 82589171), each with their supporting sequence IDs, based on which the locations of the corresponding basic copy-pasted segments (file name and line numbers) can be identified.

3.1.3 Composing Larger Copy-pasted Segments

Since every sequence fed to the mining algorithm represents a basic block, a basic copy-pasted segment may only be a part of a larger copy-pasted segment. Therefore, it is necessary to combine a basic copy-pasted segment with its neighbors to construct a larger one, if possible.

The composition procedure is very straightforward. CP-Miner maintains a candidate set of copy-paste groups, which initially includes all of the basic copy-pasted segments that survive the pruning procedure described in Section 3.1.4. For each copy-paste group, CP-Miner checks their neighboring code segments to see if they also form a copy-paste group. If so, the two groups are combined together to form a larger one. This larger copy-paste group is checked against the pruning procedure. If it can survive the pruning process, it is added to the candidate set and the two smaller ones are removed. Otherwise, the two smaller ones still remain in the set and are marked as “non-expandable”. CP-Miner repeats this process until all groups in the candidate set are non-expandable.

3.1.4 Pruning False Positives

It is possible that copy-pasted segments discovered by the mining algorithm or the composition process may contain false positives. The main cause of false positives is the tokenization of identifiers (variable/function/type) in order to tolerate identifier-renaming in copy-paste. Since identifiers of the same type are mapped into the same token, it is possible to identify false copy-pasted segments. For example, all statements similar to $x = y + z$ would have

the same hash value, which can introduce many false positives. To prune false positives, CP-Miner has applied several techniques to both of basic and composed copy-pasted segments. The pruning techniques include:

(1) Pruning unmappable segments: This technique is used to prune false positives introduced by the tokenization of identifiers. This is based on the observation that if a programmer copy-pastes a code segment and then renames an identifier, he/she would most likely rename this identifier in all its occurrences in the new copy-pasted segment. Therefore, we can build an identifier mapping that maps old names in one segment to their corresponding new ones in the other segment that belongs to the same copy-paste group. In the example shown in Figure 2, variable *prom_phys_total* is changed into *prom_prom_taken* (except the bug on line 117).

A mapping scheme is consistent if there are very few conflicts that map one identifier name to two or more different new names. If no consistent identifier mapping can be established between a pair of copy-pasted segments, they are likely to be false positives.

To measure the amount of conflict, CP-Miner uses a metric called *ConflictRatio*, which records the conflict ratio for an identifier mapping between two candidate copy-pasted segments. For example, if a variable *A* from segment 1 is changed into *a* in 75% of its occurrences in segment 2 but 25% of its occurrences is changed into other variables, the *ConflictRatio* of mapping $A \rightarrow a$ is 25%. The *ConflictRatio* for the whole mapping scheme between these two segments are the weighted sum of *ConflictRatio* of the mapping for each unique identifier. The weight for an identifier *A* in a given code segment is the fraction of total identifier occurrences that are occurrences of *A*. If *ConflictRatio* for two candidate copy-pasted segments is higher than a predefined threshold, these two code segments are filtered as false positives. In our experiments, we set the threshold to be 60%.

(2) Pruning tiny segments: Our mining algorithm may find tiny copy-pasted segments that consist of only 1-2 simple statements. If such a tiny segment cannot be combined with neighbors to compose a larger segment, it is removed from the copy-paste list. This is based on the observation that copy-pasted segments are usually not very small because programmers cannot save much effort in copy-pasting a simple tiny code segment.

CP-Miner uses the number of tokens to measure the size of a segment. This metric is more appropriate than the number of statements, because the length of statements is highly variable. If a single statement is very complicated with many tokens, it is still possible for programmers to copy-paste it.

To prune tiny segments, CP-Miner uses a tunable parameter called *min_size*. If the number of tokens in a pair of copy-pasted segments is fewer than *min_size*, this pair is removed.

(3) **Pruning overlapped segments:** If a pair of candidate copy-pasted segments overlap with each other, they are also considered false positives. CP-Miner stops extending the pair of copy-pasted segments once they overlap. For some program structures such as the *switch* statement that contain many pairs of self-similar segments, pruning overlapped segments can avoid most of the false positives in *switch* statements.

(4) **Pruning segments with large gaps:** Besides the mining procedure for basic copy-pasted segments, the gap constraint is also applied to composed ones. When two neighboring segments are combined, the maximum gap of the newly composed large segment may become larger than a predefined threshold, *max_total_gap*. If this is true, the composition is invalid. So the newly composed one is not added into the candidate set and the two smaller ones are marked as non-expandable in the set.

Of course, even after such rigorous pruning, false positives may still exist. However, we have manually examined 100 random copy-pasted segments reported by CP-Miner for Linux, and only a few false positives (8) are found. We can only manually examine each identified copy-pasted segment because there are no traces that record programmers' copy-paste operations during the development of the software.

3.1.5 Computational Complexity of CP-Miner

CP-Miner can extract copy-pasted code directly from a single software with total complexity of $O(n^2)$ in the worst case (where n is the number of lines of code), and the optimizations further improve its efficiency in practice. For example, CP-Miner can identify more than 150,000 copy-pasted segments from 3–4 million lines of code in less than 20 minutes as shown in our results in Section 5.3. In CP-Miner, we break all of the large basic blocks into small blocks with at most 30 statements before feeding to the mining algorithm. Therefore, the search tree is at most with depth 30. With this constraint of search tree, the mining complexity of CP-Miner is $O(n^2)$ in the worst case. Furthermore, the optimizations described in Section 2.2 make it more efficient in both time and space overheads than the worst case.

3.2 Detecting Copy-paste Related Bugs

As we have mentioned in Section 1, the main cause of copy-paste related bugs is that programmers forget to modify identifiers consistently after copy-pasting. Once we get the mapping relationship between identifiers in a pair of copy-pasted segments (see Section 3.1.4), we can find the inconsistency and report these copy-paste related bugs. Table 1 shows the identifier mapping for the example described in Section 1.

For an identifier that appears more than once in a copy-pasted segment, it is consistent when it always maps to the same identifier in the other segment. Similarly, it is inconsistent when it maps itself to multiple identifiers. In

Identifiers in segment I (line 92-99)	Identifiers in segment II (line 111-118)
iter (9)	iter (9)
num.reg (1)	num.reg (1)
prom.phys.total (4)	prom.prom.taken (3); prom.phys.total (1)
prom.reg.memlist (2)	prom.reg.memlist (2)

Table 1: Identifier mapping in the example in Figure 1 (the number after each identifier indicates the number of occurrences).

Table 1, we can see that *prom.phys.total* is mapped inconsistently, because it maps to *prom.prom.taken* three times and *prom.phys.total* once. All the other variable mappings are consistent.

Unfortunately, inconsistency does not necessarily indicate a bug. If the amount of inconsistency is high, it may indicate that the code segments are not copy-pasted. Section 3.1.4 describes how we prune unmapable copy-pasted segments based on this observation.

Therefore, the challenge is to decide when an inconsistency is likely to be a bug instead of a false positive of copy-paste. To address this challenge, we need to consider the programmers' intention. Our bug detection method is based on the following observation: if a programmer makes a change in a copy-pasted segment, the changed identifier is unlikely to be a bug. But if he/she changes an identifier in most places but forgets to change it in a few places, the unchanged identifier is likely to be a bug. In other words, "forget-to-change" is more likely to be a bug than an intentional "change". For example, if in some cases, an identifier *A* is mapped into *a* and in other cases it is mapped into *a'* (both *a* and *a'* are different from *A*), it is unlikely to be a bug because programmers *intentionally* change *A* to other names. On the other hand, if *A* is changed into *a* in most cases but remains unchanged only in a few cases, the unchanged places are likely to be bugs.

Based on the above observation, CP-Miner reexamines each non-expandable copy-paste group after running through the pruning and composing procedures. For each pair of copy-pasted segments, it uses a metric called *UnchangedRatio* to detect bugs in an identifier mapping. We define

$$UnchangedRatio = \frac{NumUnchanged}{NumTotal}$$

where *NumUnchanged* means the number of occurrences that a given identifier is unchanged, and *NumTotal* means the number of total occurrences of this identifier in a given copy-pasted segment. Therefore, the lower the *UnchangedRatio*, the more likely it is a bug, unless *UnchangedRatio* = 0, which means that all of its occurrences have been changed. Note that *UnchangedRatio* is different from *ConflictRatio*. The former only measures the ratio of unchanged occurrences, whereas the latter measures the ratio of conflicts. In the example shown on Table 1, *UnchangedRatio* for *prom.phys.total* is 0.25, whereas all other identifiers have *UnchangedRatio* = 1.

CP-Miner uses a threshold for *UnchangedRatio* to detect bugs. If *UnchangedRatio* for an identifier is not zero

and not larger than the threshold, the unchanged places are reported as bugs. When CP-Miner reports a bug, the corresponding identifier mapping information is also provided to programmers to help in debugging. In the example shown on Table 1, identifier *prom_phys_total* on line 117 is reported as a bug.

It is possible to further extend CP-Miner’s bug detection engine. For example, it might be useful to exploit variable correlations. Assume variable *A* always appears in close range to another variable *B*, and *a* always appears very close to *b*. So if in a pair of copy-pasted segments, *A* is renamed to *a*, *B* then should be renamed to *b* with high confidence. Any violation of this rule may indicate a bug. But the current version of CP-Miner has not exploited this possibility. It remains as our future work.

4 Methodology

We have evaluated the effectiveness of CP-Miner with large software including Linux, FreeBSD, Apache web server and PostgreSQL. The number of files (only C files) and the number of lines of code (LOC) for the software are shown in Table 2.

Software	version	#files	#LOC
Linux	2.6.6	6,497	4,365,124
FreeBSD	5.2.1	7,114	3,299,622
Apache	2.0.49	479	223,886
PostgreSQL	7.4.2	553	458,058

Table 2: Software evaluated in our experiments.

We set the thresholds used in CP-Miner as following. The minimum copy-pasted segment size *min_size* is 30 tokens. We also vary the gap constraints: (1) when *max_gap* = 0, CP-Miner only identifies copy-pasted code with identifier-renaming; (2) when *max_gap* = 1 and *max_total_gap* = 2, it means that CP-Miner allows copy-pasted segments with insertion and deletion of one statement between any two consecutive statements, and a total of two statement insertions and deletions in the whole segment. Without specifying, we use setting (2) by default.

We define *CP_Coverage* to measure the percentage of copy-paste in given software (or a given module):

$$CP_Coverage = \frac{\#LOC \text{ in copy-pasted segments}}{\#LOC \text{ in the software or the module}} \times 100\%$$

In our experiments, we also compare CP-Miner with a recently proposed tool called *CCFinder* [24]. Similar to our tool, CCFinder also tokenizes identifiers, keywords, constant, operators, etc. But different from our tool, it uses a suffix tree algorithm instead of a data mining algorithm. Therefore, it cannot tolerate statement insertions and deletions in copy-pasted code. Our results show that CP-Miner detects 17–52% more copy-pasted code than CCFinder. In addition, CCFinder does not filter incomplete, tiny copy-pasted segments which are very likely to be false positives. CCFinder does not detect copy-paste related bugs, so we cannot compare this functionality between them.

In our experiments, we run CP-Miner and CCFinder on an Intel Xeon 2.4GHz machine with 2GB memory.

5 Evaluation Results of CP-Miner

We first present the evaluation results of CP-Miner in this section, including the number of copy-pasted segments, the number of detected copy-paste related bugs, CP-Miner overhead, comparison with CCFinder, and effects of threshold setting. The statistical results of copy-paste characteristics in Linux and FreeBSD will be presented in Section 6.

5.1 Overall Results

Detecting Copy-pasted Code CP-Miner has found a significant number of copy-pasted segments in the evaluated software. In this software, copy-pasted code makes up 17.7–22.3% of the code base. Table 3 shows the numbers of copy-pasted segments and *CP_Coverage*. As shown in this table, in Linux and FreeBSD, there are more than 100,000 and 120,000 copy-pasted segments without any statement insertion (*max_gap* = 0), which accounts for about 15% of the source code. We have manually examined 100 random pairs of copy-pasted segments from all potential copy-pasted segments in Linux (with *max_gap* = 1), and found a few (only 8) false positives. The large number of copy-pasted segments motivates a support in software development environments such as Microsoft Visual Studio to maintain copy-pasted code.

Software	<i>max_gap</i> = 0		<i>max_gap</i> = 1	
	#Segments	<i>CP_Coverage</i>	#Segments	<i>CP_Coverage</i>
Linux	122,282	15.3%	198,605	22.3%
FreeBSD	101,699	14.9%	153,230	20.4%
Apache	4,155	13.1%	6,196	17.7%
PostgreSQL	12,105	16.5%	16,662	22.2%

Table 3: The number of copy-pasted segments and *CP_Coverage*

Our results also show that a large percentage (30–50%) of copy-pasted segments have statement insertions and modifications. For example, when *max_gap* is 1, CP-Miner finds 62.4% more copy-pasted segments in Linux. In FreeBSD, the *CP_Coverage* increases from 14.9% to 20.4% when *max_gap* is relaxed from 0 to 1. These results show that previous tools including CCFinder that cannot tolerate statement insertions and modifications would miss a lot of copy-paste.

By increasing *max_gap* from 1 to 2 or higher, we can further relax the gap constraint. Due to space limitation, we do not show those results here. Also the number of false positives will increase with *max_gap*. Our manual examination results with the Linux file system module indicate that false positives are low with *max_gap* = 1, and relatively low with *max_gap* = 2.

Detecting Copy-paste Related Bugs CP-Miner has also reported many copy-paste related errors in the evaluated software. Since the errors reported by CP-Miner may not be bugs, we verify each reported error manually and then report to the corresponding developer community those errors that we suspect to be bugs with high confidence. The

Software	errors reported	bugs verified	careless programming	false alarms		
				(1)	(2)	(3)
Linux	421	28	21	151	41	57
FreeBSD	443	23	8	307	41	30
Apache	17	5	0	3	1	6
PostgreSQL	74	2	0	13	10	43

Table 4: Errors reported by CP-Miner (*UnchangedRatio* threshold = 0.4) and bugs verified by us with high confidence, some of which are confirmed and fixed by corresponding developers after we reported. The false alarms include three categories: (1) incorrectly matched segments, (2) exchangeable orders, and (3) others. The first two categories can be pruned, which remains as our immediate future work.

numbers of errors found by CP-Miner and verified bugs are shown on Table 4. The results are achieved by setting the *UnchangedRatio* threshold to be 0.4.

Both Linux and FreeBSD have many copy-paste related bugs. So far, we have verified 28 and 23 bugs in the latest versions of Linux and FreeBSD. Most of these bugs had never been reported before. We have reported these bugs to the kernel developer communities. Recently, five Linux bugs have been confirmed and fixed by kernel developers, and the others are still in the process of being confirmed.

Since Apache and PostgreSQL are much smaller compared to Linux and FreeBSD, CP-Miner found much fewer copy-paste related bugs. We have verified 5 bugs for Apache and 2 bugs for PostgreSQL with high confidence. One bug in Apache was immediately fixed by the Apache developers after we reported it to them.

In addition to those bugs verified, we also find many “potential bugs” (21 in Linux and 8 in FreeBSD) that are not bugs by coincidence but might become bugs in the future. We call this type of errors “careless programming”. Similar to the bugs verified, these errors also forget to change some identifiers consistently at a few places. Fortunately, by coincidence, the new identifiers and the old ones happen to have the same values. However, if such implicit assumptions are violated in future versions of the software, it would lead to bugs that are hard to detect.

5.2 False Alarms

Table 4 also shows the number of false alarms reported by CP-Miner. These false alarms are mostly caused by the following two major reasons and can be further pruned in our immediate future work:

(1) Incorrectly matched copy-pasted segments: In some copy-pasted segments that contain multiple “*case*” or “*if*” blocks, there are many possible combinations for these contiguous copy-pasted blocks to compose larger ones. Since CP-Miner simply follows the program order to compose larger copy-pastes, it is likely that a wrong composition might be chosen. As a result, identifiers are compared between two incorrectly matched copy-pasted segments, which results in false alarms.

These false alarms can be pruned if we use more semantic information of the identifiers in these segments. The segments with a number of “*case/if*” blocks usually contain a lot of constant identifiers, but our current CP-Miner treats them as normal variable names. If we use the infor-

mation of these constants to match “*case/if*” blocks when composing larger copy-pasted segments, it can reduce the number of incorrectly matched segments and most of such false alarms can be pruned.

(2) Exchangeable orders: In a copy-paste pair, the orders of some statements or expressions can be switched. For example, a segment with several similar statements such as “*a1=b1; a2=b2;*” is the same as “*a2=b2; a1=b1;*”. The current version of CP-Miner simply compares the identifiers in a pair of copy-pasted segments in strict order and therefore a false alarm might be reported. In Linux, 41 false alarms are caused by such exchangeable orders.

These false alarms can be pruned if we relax the strict order comparison by further checking whether the corresponding “changed” identifiers are in the neighboring statements/expressions.

5.3 Time and Space Overheads

CP-Miner can identify copy-pasted code in large software very efficiently. The execution time of CP-Miner is shown in Table 5. CP-Miner takes 11–20 minutes to identify 101,699–198,605 copy-pasted segments in Linux and FreeBSD, each with 3–4 million lines of code. It takes less than 1 minute to detect copy-pasted segments in Apache and PostgreSQL with more than 200,000 lines of code.

CP-Miner is also space-efficient. For example, it takes less than 530MB to find copy-pasted code in Linux and FreeBSD. For Apache and PostgreSQL, CP-Miner consumes 27–57 MB of memory.

Software	<i>max_gap</i> = 0		<i>max_gap</i> = 1	
	Time(s)	Space(MB)	Time(s)	Space(MB)
Linux	770	438	1164	527
FreeBSD	615	334	1155	459
Apache	14	27	15	30
PostgreSQL	32	44	38	57

Table 5: Execution time and memory space of CP-Miner

5.4 Comparison with CCFinder

We have compared CP-Miner with CCFinder [24]. CCFinder has execution time similar to that of CP-Miner, but CP-Miner discovers much more copy-pasted segments. In addition, CCFinder does not detect copy-paste related bugs. As we explained in Section 4, CCFinder allows identifier-renaming but not statement insertions. In addition, pruning in CCFinder is not so rigorous as CP-Miner. For example, CCFinder reports incomplete statements in copy-pasted segments, which is unlikely in practice. After pruning the incomplete statements, many small copy-

Software	CCFinder	CP-Miner
Linux	14.7%(19.8%)	22.3%
FreeBSD	14.5%(19.6%)	20.4%
Apache	11.8%(15.3%)	17.7%
PostgreSQL	18.5%(23.8%)	22.2%

Table 6: *CP_Coverage* comparison between CP-Miner and CCFinder. For CCFinder, the first number is the result after pruning those incomplete, small segments, and the second number in parentheses is the result before pruning.

pasted segments consist of less than 30 tokens, which are too simple to be worth copying.

CP-Miner can identify 17–52% more copy-pasted code than CCFinder because CP-Miner can tolerate statement insertions and modifications. Table 6 compares the *CP_Coverage* identified by CP-Miner and CCFinder. The results with CP-Miner are achieved using the default threshold setting (*min_size* = 30 and *max_gap* = 1). For fair comparison, we also filter those incomplete, small segments from CCFinder’s output.

5.5 Effects of Threshold Settings

Segment Size Threshold Figure 3 shows the effect of segment size threshold *min_size* on *CP_Coverage*. As expected, *CP_Coverage* decreases when *min_size* increases because more copy-pasted segments are pruned. The results also show that the decrement slowdowns when *min_size* is in the range of 30–100 tokens, which indicates that not too many copy-segments’ sizes fall in this range. This implies that segments with fewer than 30 tokens are very likely to be false positives, whereas those with more than 40 tokens are very likely to be copy-paste.

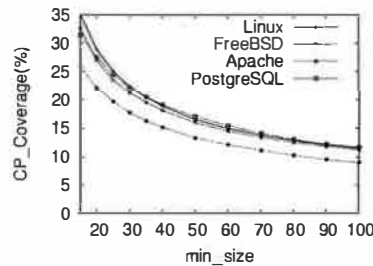


Figure 3: Effects of *min_size* on *CP_Coverage*

Unchanged Ratio Threshold Figure 4 shows the effect of unchanged ratio threshold on the number of bugs reported. Since *UnchangedRatio* ≥ 0.5 means that most of the identifiers are not changed after copy-pasting, these unchanged identifiers are unlikely “forget-to-change” and so it cannot indicate a copy-paste related error. Therefore, we only show the errors with *UnchangedRatio* threshold less than 0.5.

As expected, more errors are reported by CP-Miner when the *UnchangedRatio* threshold increases. Specifically, the number of errors reported increases gradually

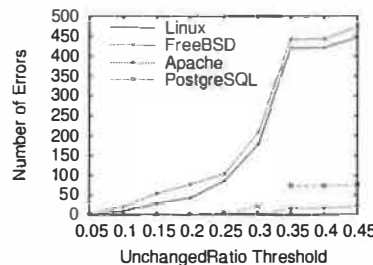
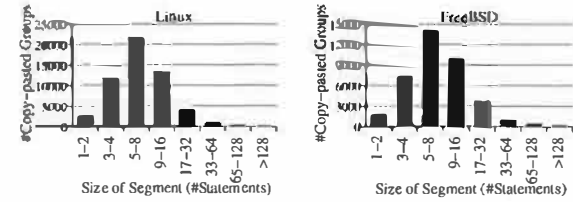
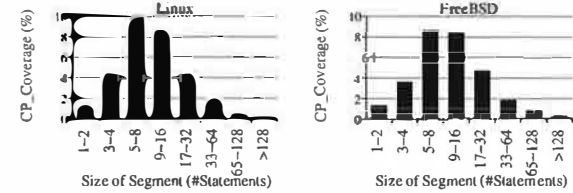


Figure 4: Effects of *UnchangedRatio* threshold on errors reported



(a) # of copy-paste groups with various segment sizes (# of statements)



(b) The *CP_Coverage* with various segment sizes (# of statements)

Figure 5: Size distribution of copy-pasted segments. Due to the overlap of copy-pasted segments that have different segment sizes; and also belong to different copy-paste groups, the sum of all *CP_Coverage* does not equal to the overall *CP_Coverage*.

when the threshold is less than 0.25, and then increases sharply when the threshold $\in (0.25, 0.35)$. We found that most of the errors with high *UnchangedRatio* turn out to be false alarms during our verification. For example, CP-Miner reports many errors where only 1 out of 3 identifiers is unchanged (*UnchangedRatio* = 0.33). However, it cannot strongly support that it is a copy-paste related bug. In order to prune such false alarms, we can further analyze the identifiers in the context of the copy-pasted segments (e.g., the whole function). We leave this improvement as our future work.

6 Statistics of Copy-paste in OS code

This section presents the statistical results on copy-paste characteristics in large software. Our results include the distribution of copy-pasted segments across different group sizes, segment sizes, granularity, amount of changes, modules, and versions.

6.1 Copy-paste Size and Granularity

Figure 5 illustrates the distribution of copy-pasted segments with different sizes (in terms of the number of statements). The results show that most (60–64%) copy-pasted segments are not very large, with only 5–16 statements. Only a few (0.2–5.0%) copy-pasted segments have more than 64 statements. In particular, Figure 5(a) shows that most (35–40%) copy-paste groups contain 5–8 statements in each segment. Figure 5(b) shows similar characteristics: copy-pasted segments with 5–8 statements cover about 7–10% of the source code.

Figure 6 shows the distribution of copy-paste group size. About 60% of copy-paste groups contain only two segments, which indicates that there are only two copies (original and replicated) for most copy-pasted code. But still, a lot of code is replicated more than once.

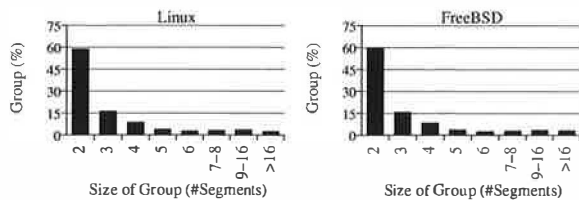


Figure 6: Copy-paste group size distribution in terms of the number of segments in each group. Each bar represents the percentage of copy-paste groups that contain the corresponding number of segments.

Software	basic block	function
Linux	17,818 (9.0%)	26,744 (13.5%)
FreeBSD	13,999 (9.1%)	17,254 (11.3%)

Table 7: Distribution of copy-paste granularity: numbers and percentages of copy-pasted segments at different granularity. Note here the percentage is not *CP_Coverage*. It is calculated by comparing to the total number of copy-pasted segments.

Total 6.3–6.7% of copy-pasted segments are copy-pasted more than 8 times. If a bug is detected in one of the copies, it is difficult for programmers to remember fixing the bug in the other 8 or more copies. This motivates a tool that can automatically fix other copy-pasted segments once a programmer fixes one segment.

Table 7 shows the number of copy-pasted segments at basic-block and function granularity. Our results show that 9% of copy-pasted segments are basic blocks, which indicates that programmers seldom copy-paste basic blocks because most of them are too simple to worth it.

More interestingly, there are 13.5% of copy-pasted segments with whole functions in Linux and 11.3% in FreeBSD. The reason is that many functions provide similar functionalities, such as reading data from different types of devices. Those functions can be copy-pasted with modifications such as replacing data types of parameters. This motivates some refactoring tools [23] to better maintain these copy-pasted functions.

6.2 Modifications in Copy-pasted Segments

Figure 7 shows how many identifiers are changed in copy-pasted segments. Since in some cases there are more than two segments in each copy-paste group, we only present the distribution in the best case: comparing the most similar pair of segments from each copy-paste group. Each bar includes two parts: one with no statement insertion and the other with one statement insertion.

The results indicate that 65–67% of copy-pasted segments require identifier renaming. For example, in Linux, 27% copy-pasted segments are identical, and 8% segments are almost identical with only one statement inserted. The rest 65% of the copy-pasted segments in Linux rename at least one identifier. Such results motivate a tool to support consistently renaming identifiers in copy-pasted code.

The results in Figure 7 also show that about 23–27% of copy-pasted segments contain at least one statement insertion, deletion, and modification (*Gap*=1). It indicates that it is important for copy-paste detection tools to tolerate such statement modifications.

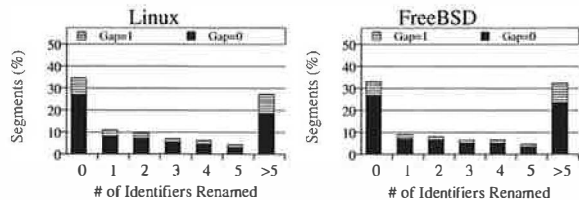
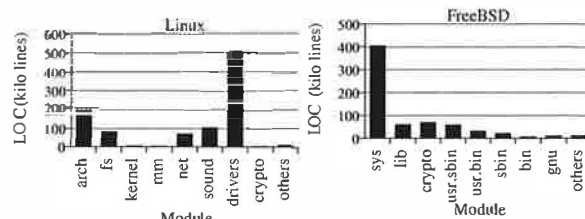
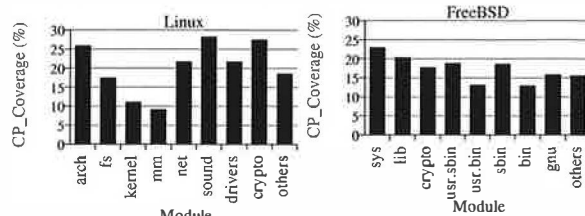


Figure 7: Distribution of identifiers changed in copy-pasted segments. Each bar represents the percentage of segments that have the corresponding number of renamed identifiers. Each bar has two parts: “*Gap* = 0” and “*Gap* = 1” represent the copy-pasted segments with no and one statement modifications, respectively.



(a) The number of copy-pasted lines in different modules



(b) *CP_Coverage* in different modules

Figure 8: Copy-pasted code in different modules.

6.3 Copy-pasted Code across Modules

Different modules have different copy-paste characteristics. In this subsection, we analyze copy-pasted code across different modules in operating system code. We split Linux into 9 categories: arch (platform specific), fs (file system), kernel (main kernel), mm (memory management), net (networking), sound (sound device drivers), drivers (device drivers other than networking and sound device), crypto (cryptography), and others (all other code). For FreeBSD, modules are also split into 9 categories: sys (kernel sources), lib (system libraries), crypto (cryptography), usr.sbin (system administration commands), usr.bin (user commands), sbin (system commands), bin (system/user commands), gnu, and others.

Figure 8 shows the number and *CP_Coverage* of copy-pasted segments in different modules. The *CP_Coverage* is computed based on the size of each corresponding module, instead of the entire software.

Figure 8 (a) shows that most copy-pasted code in Linux and FreeBSD is located in one or two main modules. For example, modules “drivers” and “arch” account for 71% of all copy-pasted code in Linux, and module “sys” accounts for 60% in FreeBSD. This is because many drivers are similar, and it is much easier to modify a copy-paste of another driver than writing one from scratch.

Figure 8 (b) shows that a large percentage (20–28%) of

the code in Linux is copy-pasted in the “arch” module, the “crypto” module, and the device driver modules including “net”, “sound”, and “drivers”. The “arch” module has a lot of copy-pasted code because it has many similar sub-modules for different platforms. The device driver modules contain a significant portion of copy-pasted code because many devices share similar functionalities. Additionally, “crypto” is a very small module (less than 10,000 LOC), but the main cryptography algorithms consist of a number of similar computing steps, so it contains a lot of copy-pasted code. Our results indicate that more attention should be paid to these modules because they are more likely to contain copy-paste related bugs.

In contrast, the modules “mm” and “kernel” contain much less copy-pasted code than others, which indicates that it is rare to reuse code in kernels and memory management modules.

6.4 Evolution of Copy-paste

Figure 9 shows that the copy-pasted code increases as the operating system code evolves. For example, Figure 9(a) shows that as Linux’s code size increases from 141,000 to 4.4 million lines, copy-pasted code also keeps increasing from 23,000 to 975,000 lines through version 1.0 to 2.6.6.

In terms of *CP_Coverage*, the percentage of copy-pasted code also steadily increases along software evolution. For example, Figure 9(a) shows that *CP_Coverage* in Linux increases from 16.2% to 22.3% from version 1.0 to 2.6.6, and Figure 9(b) shows that *CP_Coverage* in FreeBSD increases from 17.5% to 21.7% from version 2.0 to 4.10. However, the *CP_Coverage* remains relatively stable over the recent several versions for both Linux and FreeBSD. For example, the *CP_Coverage* for FreeBSD has been staying around 21–22% since version 4.0.

7 Related Work

In this section, we briefly discuss closely related work that has not been described in earlier sections.

7.1 Detecting Copy-Pasted Code

Several studies have been conducted on detection of copy-pasted code. The techniques used include: line-by-line [6], token-by-token [24, 33], fingerprinting [21], visualization [11, 13], abstract syntax tree [7, 27], and dependence graph [26, 28].

Dup [6] finds all pairs of matching *parameterized* code fragments. A code fragment matches another if both fragments are contiguous sequences of source lines with some consistent identifier mapping scheme. Because this approach is line-based, it is sensitive to lexical aspects like the presence or absence of new lines. In addition, it does not find non-contiguous copy-pastes. CP-Miner does not have these shortcomings.

Johnson [21] proposed using a fingerprinting algorithm on a substring of the source code. In this algorithm, calculated signatures per line are compared in order to iden-

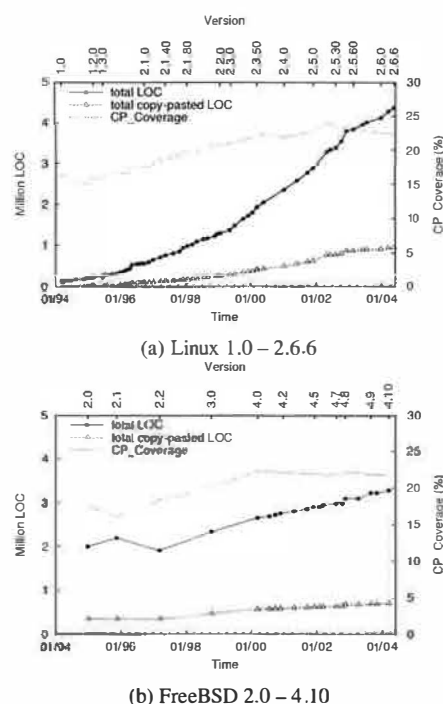


Figure 9: Copy-pasted code in Linux and FreeBSD through various versions. The x-axis (version number) is drawn in time scale with the corresponding release time. The versions of Linux we analyze are through 1.0 to the current version 2.6.6. The versions of FreeBSD include the main branch through 2.0 to 4.10.

tify matched substrings. As with line-based techniques, this approach is sensitive to minor modifications made in copy-pasted code.

Some graphical tools were proposed to understand code similarities in different programs (or in the same program) visually. *Dotplots* [11] of source code can be constructed by tokenizing the code into lines and placing a dot in coordinates (i, j) on a 2-D graph, if the i^{th} input token matches j^{th} input token. Similarly, *Duploc* [13] provides a scatter plot visualization of copy-pastes (detected by string matching of lines) and also textual reports that summarize all discovered sequences. Both *Dotplots* and *Duploc* only support line granularity. In addition, they can only detect identical duplicates and do not tolerate renaming, insertions, and deletions.

Baxter et al. [7] proposed a tool that transforms source code into abstract-syntax trees (AST), and detects copy-paste by finding identical subtrees. Similar to other tools, it is not tolerant to modifications in copy-pasted segments. In addition, it may introduce many false positives because two code segments with the same syntax subtrees are not necessarily copy-pastes.

Komondoor et al. [26] proposed using program dependence graph (PDG) and program slicing to find isomorphic subgraphs and code duplication. Although this approach is successful at identifying copies with reordered statements, its running time is very long. For example, it takes 1.5 hours to analyze only 11,540 lines of source code from

bison, much slower than CP-Miner. Another slow PDG-based approach is found in [28].

Mayrand et al. [31] used an Intermediate Representation Language to characterize each function in the source code and detect copy-pasted function bodies that have similar metric values. This tool does not detect copy-paste at other granularity such as segment-based copy-paste, which occurs more frequently than function-based copy-paste as shown in our results.

Some copy-paste detection techniques are too coarse-grained to be useful for our purpose. *JPlag* [33], *Moss* [35], and *sif* [30] are tools to find similar programs among a given set. They have been commonly used to detect plagiarism. Most of them are not suitable for detecting copy-pasted code in a single large program.

Kontogiannis et al. [27] built an abstract pattern matching tool to identify probable matches using Markov models. This approach does not find copy-pasted code. Instead, it only measures similarity between two programs.

7.2 Detecting Software Bugs

Many tools have been proposed for detecting software bugs. One approach is dynamic checking that detects bugs during execution. Examples of dynamic tools include Purify [19], Valgrind [36], DIDUCE [18], Eraser [34], and CCured [12]. Dynamic tools have more accurate information but may introduce overheads during execution. Moreover, they can only find bugs on the execution paths. Most dynamic tools cannot detect bugs in operating systems.

Another approach is to perform checks statically. Examples of this approach include explicit model checking [15, 32, 37] and program analysis [8, 14, 17]. Most static tools require significant involvement of programmers to write specifications or annotate programs. But the advantage of static tools is that they add no overhead during execution, and it can find bugs that may not occur in the common execution paths. A few tools do not require annotations, but they focus on detecting different types of bugs, instead of copy-paste related bugs.

Our tool, CP-Miner, is a static tool that can detect copy-paste related bugs, *without any annotation requirement from programmers*. CP-Miner complements other bug detection tools because it is based on a different observation: finding bugs caused by copy-paste. Some copy-paste related bugs can be found by previous tools if they lead to buffer overflow or some obvious memory corruption, but many of them, especially those semantic ones, cannot be found by previous tools.

Our work is motivated by and related to Engler et al.'s empirical analysis of operating systems errors [10]. Their study gave an overall error distribution and evolution analysis in operating systems, and found that copy-paste is one of the major causes for bugs. Our work presents a tool to detect copy-pasted code and related bugs in large software including operating system code. Many of these bugs such as the one in Figure 1 cannot be detected by their tools.

8 Conclusions

This paper presents a tool called CP-Miner² that uses data mining techniques to efficiently identify copy-pasted code in large software including operating systems, and also detects copy-paste related bugs. Specifically, it takes less than 20 minutes for CP-Miner to identify 190,000 and 150,000 copy-pasted segments that account for 20–22% of the source code in Linux and FreeBSD. Moreover, CP-Miner has detected 28 and 23 copy-paste related bugs in the latest versions of Linux and FreeBSD, respectively. Compared to CCFinder [24], CP-Miner finds 17–52% more copy-pasted segments because it can tolerate statement insertions and modifications in copy-paste. In addition, we have shown some interesting characteristics of copy-pasted codes in Linux and FreeBSD, including distribution of copy-paste across different segment sizes, group sizes, granularity, modules, amount of modifications, and software evolution.

Our results indicate that maintaining copy-pasted code would be very useful for programmers because it is commonly used in large software such as operating system code, and it can easily introduce hard-to-detect bugs. We hope our study motivates software development environments such as Microsoft Visual Studio to provide functionality to maintain copy-pasted code and automatically detect copy-paste related bugs.

Even though CP-Miner focuses only on “forget-to-change” bugs caused by copy-paste, copy-paste can introduce many other types of bugs. For example, after copy-paste operation, the programmer forgets to add some statements that are specific to the new copy-pasted segment. However, such bugs are hard to detect because it relies on semantic information. It is impossible to guess what the programmer would want to insert or modify. Another type of copy-paste related bugs is caused by programmers forgetting to fix a known bug in all copy-pasted segments. They only fix one or two segments but forget to change it in the others. Our tool CP-Miner can detect simple cases of this type of errors. But if the fix is too complicated, CP-Miner would miss the bug because the modified code segment becomes too different from the others to be identified as copy-paste. To solve this problem more thoroughly, it would require support from software development environments such as Microsoft Visual Studio.

9 Acknowledgements

The authors would like to thank the shepherd, Andrew Myers, the anonymous reviewers, and James Larus (Microsoft Research Lab) for their invaluable feedback. We appreciate Professor Jiawei Han and his group for their CloSpan mining algorithm. We would also like to thank Cigdem Sengul for her help with the initial investigation of our project. This research is supported by IBM Faculty

²CP-Miner will be released to the research community.

Award, NSF CNS-0347854 (career award), NSF CCR-0305854 grant and NSF CCR-0325603 grant. Our experiments were conducted on equipment provided through the IBM SUR grant.

REFERENCES

- [1] Linux kernel mailing list. <http://lkml.org>.
- [2] R. Agrawal and R. Srikant. Mining sequential patterns. In *Eleventh International Conference on Data Engineering*, 1995.
- [3] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.
- [4] A. Aiken. Moss: A system for detecting software plagiarism. <http://www.cs.berkeley.edu/~aiken/moss.html>.
- [5] B. S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, 24:49–57, 1992.
- [6] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the Second Working Conference on Reverse Engineering*, page 86. IEEE Computer Society, 1995.
- [7] I. D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, page 368. IEEE Computer Society, 1998.
- [8] J.-D. Choi, K. Lee, A. Loginov, R. O’Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceeding of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 258–269. ACM Press, 2002.
- [9] A. Chou, B. Chelf, D. R. Engler, and M. Heinrich. Using meta-level compilation to check FLASH protocol code. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating System*, pages 59–70. ACM Press, 2000.
- [10] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In *Symposium on Operating Systems Principles*, pages 73–88, 2001.
- [11] K. W. Church and J. I. Helfman. Dotplot: A program for exploring self-similarity in millions of lines of text and code. *Journal of Computational and Graphical Statistics*, 1993.
- [12] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 232–244. ACM Press, 2003.
- [13] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of International Conference on Software Maintenance*, pages 109–118. IEEE, 1999.
- [14] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 237–252. ACM Press, 2003.
- [15] D. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 57–72. ACM Press, 2001.
- [16] S. Grier. A tool that detects plagiarism in Pascal programs. In *Proceedings of the 12th SIGCSE Technical Symposium on Computer Science Education*, pages 15–20. ACM Press, 1981.
- [17] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 69–82. ACM Press, 2002.
- [18] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, May 2002.
- [19] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, pages 158–185, Dec 1992.
- [20] H. T. Jankowitz. Detecting plagiarism in student Pascal programs. *Computer Journal*, 31(1):1–8, 1988.
- [21] J. H. Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the conference of the Centre for Advanced Studies on Collaborative research*, Toronto, Ontario, Canada, October 1993.
- [22] J. H. Johnson. Substring matching for clone detection and change tracking. In *Proceedings of the International Conference on Software Maintenance*, pages 120–126. IEEE Computer Society, 1994.
- [23] R. E. Johnson and W. F. Opdyke. Refactoring and aggregation. In *Object Technologies for Advanced Software, First JSSST International Symposium*, volume 742, pages 264–278. Springer-Verlag, 1993.
- [24] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilingualistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.
- [25] C. Kapser and M. W. Godfrey. Toward a taxonomy of clones in source code: A case study. *Evolution of Large-scale Industrial Software Applications (ELISA)*, Sept 2003.
- [26] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *8th International Symposium on Static Analysis (SAS)*, 2001.
- [27] K. Kontogiannis, M. Galler, and R. DeMori. Detecting code similarity using patterns. *Working Notes of the Third Workshop on AI and Software Engineering: Breaking the Toy Mold (AISE)*, 1995.
- [28] J. Krinke. Identifying similar code with program dependence graphs. In *Eighth Working Conference on Reverse Engineering (WCRE)*, 2001.
- [29] Z. Li, Z. Chen, S. Srinivasan, and Y. Zhou. C-Miner: Mining Block Correlations in Storage Systems. In *Proceedings of the 3rd USENIX Conference on File and Storage Technology*, 2004.
- [30] U. Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 1–10, San Francisco, CA, USA, 17–21 1994.
- [31] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the 1996 International Conference on Software Maintenance*, page 244. IEEE Computer Society, 1996.
- [32] M. Musuvathi, D. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, Dec. 2002.
- [33] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, 8(11):1016–1038, Nov 2002.
- [34] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.
- [35] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 76–85. ACM Press, 2003.
- [36] J. Seward. Valgrind, an open-source memory debugger for x86-GNU/Linux. available at URL <http://developer.kde.org/~sewardj/>.
- [37] U. Stern and D. L. Dill. Automatic verification of the SCI cache coherence protocol. In *Conference on Correct Hardware Design and Verification Methods*, pages 21–34, 1995.
- [38] X. Yan, J. Han, and R. Afshar. CloSpan: Mining closed sequential patterns in large datasets. In *Proceedings of 2003 SIAM International Conference on Data Mining (SDM’03)*, San Francisco, CA, May 2003.
- [39] M. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning*, 40:31–60, 2001.

Enhancing Server Availability and Security Through Failure-Oblivious Computing

Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy,
Tudor Leu, and William S. Beebe, Jr.
*Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139*

Abstract

We present a new technique, *failure-oblivious computing*, that enables servers to execute through memory errors without memory corruption. Our safe compiler for C inserts checks that dynamically detect invalid memory accesses. Instead of terminating or throwing an exception, the generated code simply discards invalid writes and manufactures values to return for invalid reads, enabling the server to continue its normal execution path.

We have applied failure-oblivious computing to a set of widely-used servers from the Linux-based open-source computing environment. Our results show that our techniques 1) make these servers invulnerable to known security attacks that exploit memory errors, and 2) enable the servers to continue to operate successfully to service legitimate requests and satisfy the needs of their users even after attacks trigger their memory errors.

We observed several reasons for this successful continued execution. When the memory errors occur in irrelevant computations, failure-oblivious computing enables the server to execute through the memory errors to continue on to execute the relevant computation. Even when the memory errors occur in relevant computations, failure-oblivious computing converts requests that trigger unanticipated and dangerous execution paths into anticipated invalid inputs, which the error-handling logic in the server rejects. Because servers tend to have small error propagation distances (localized errors in the computation for one request tend to have little or no effect on the computations for subsequent requests), redirecting reads that would otherwise cause addressing errors and discarding writes that would otherwise corrupt critical data structures (such as the call stack) localizes the effect of the memory errors, prevents addressing exceptions from terminating the computation, and enables the server to continue on to successfully process subsequent requests. The overall result is a substantial extension of the range of requests that the server can successfully process.

1 Introduction

Memory errors such as out of bounds array accesses and invalid pointer accesses are a common source of program failures. Safe languages such as ML and Java use dynamic checks to eliminate such errors — if, for example, the program attempts to access an out of bounds array element, the implementation intercepts the attempt and throws an exception. The rationale is that an invalid memory access indicates an unanticipated programming error and it is unsafe to continue the execution without first taking some action to recover from the error.

Recently, several research groups have developed compilers that augment programs written in unsafe languages such as C with dynamic checks that intercept out of bounds array accesses and accesses via invalid pointers (we call such a compiler a *safe-C* compiler) [17, 58, 45, 36, 50, 37]. These checks use additional information about the layout of the address space to distinguish illegal accesses from legal accesses. If the program fails a check, it terminates after printing an error message.

1.1 Failure-Oblivious Computing

Note that it is possible for the compiler to automatically transform the program so that, instead of throwing an exception or terminating, it simply ignores any memory errors and continues to execute normally. Specifically, if the program attempts to read an out of bounds array element or use an invalid pointer to read a memory location, the implementation can simply (via any number of mechanisms) manufacture a value to supply to the program as the result of the read, and the program can continue to execute with that value. Similarly, if the program attempts to write a value to an out of bounds array element or use an invalid pointer to write a memory location, the implementation can simply discard the value and continue. We call a computation that uses this strategy a *failure-oblivious* computation, since it is oblivious to its failure to correctly access memory.

It is not immediately clear what will happen when a program uses this strategy to execute through a memory error. When we started this project, our hypothesis was that, for at least some programs, this continued execution would produce acceptable results. To test this hypothesis, we implemented a C compiler that generates failure-oblivious code, obtained some C programs with known memory errors, and observed the execution of failure-oblivious versions of these programs. Here is a summary of our observations:

- **Acceptable Continued Execution:** We targeted memory errors in servers that correspond to security vulnerabilities as documented at vulnerability tracking web sites [13, 12]. For all of our tested servers, failure-oblivious computing 1) eliminates the security vulnerability and 2) enables the server to successfully execute through the error to continue to serve the needs of its users.
- **Acceptable Performance:** Failure-oblivious computing entails the insertion of dynamic bounds checks into the compiled program. Previous experiments with safe-C compilers have indicated that these checks usually cause the program to run less than a factor of two slower than the version without checks, but that in some cases the program may run as much as eight to twelve times slower [58, 50]. Our results are consistent with these previous results. Note that many of our servers implement interactive computations for which the appropriate performance measure is the observed pause times for processing interactive requests. For all of our interactive servers, the application of failure-oblivious computing does not perceptibly increase the pause times.

Our conclusion is that continued execution through memory errors produces completely acceptable results for all of our servers *as long as failure-oblivious computing prevents these errors from corrupting the server's address space or data structures*.

1.2 Reason for Successful Execution

Memory errors can damage a computation in several ways: 1) they can cause the computation to terminate with an addressing exception, 2) they can cause the computation to become stuck in an infinite loop, 3) they can change the flow of control to cause the computation to generate a new and unacceptable interaction sequence (either with the user or with I/O devices), 4) they can corrupt data structures that must be consistent for the remainder of the computation to execute acceptably, or 5) they can cause the computation to produce unacceptable results.

Because failure-oblivious computing intercepts all invalid memory accesses, it eliminates the possibility that the computation may terminate with an addressing exception. It is still possible for the computation to infinite loop, but we have found a sequence of return values for invalid reads that, in practice, appears to eliminate this problem for our server programs. Our servers have simple interaction sequences — read a request, process the request without further interaction, then return the response. As long as the computation that processes the request terminates, control will appropriately flow back to the code that reads the next request and there will be no unacceptable interaction sequences. Discarding invalid writes tends to localize any memory corruption effects. In particular, it prevents an access to one data unit (such as a buffer, array, or allocated memory block) from corrupting another data unit. In practice, this localization protects many critical data structures (such as widely used application data structures or the call stack) that must remain consistent for the program to execute acceptably.

The remaining issue is the potential production of unacceptable results. Manufacturing values for reads clearly has the potential to cause a subcomputation to produce an incorrect or unexpected result. The key question is how (or even if) the incorrect or unexpected result may propagate through the remaining computation to affect the overall results of the program.

All of our initially targeted memory errors eventually boil down to buffer-overflow problems: as it processes a request, the server allocates a fixed-size buffer, then (under certain circumstances) fails to check that the data actually fits into this buffer. An attacker can exploit this error by submitting a request that causes the server to write beyond the bounds of the buffer to overwrite the contents of the stack or heap, typically with injected code that the server then executes. Such attacks are currently the most common source of exploited security vulnerabilities in modern networked computer systems [2]. Estimates place the total cost of such attacks in the billions of dollars annually [3].

Failure-oblivious computing makes a server invulnerable to this kind of attack — the server simply discards the out of bounds writes, preserving the consistency of the call stack and other critical data structures. For two of our servers the memory errors occur in computations and buffers that are irrelevant to the overall results that the server produces for that request. Because failure-oblivious computing eliminates any addressing exceptions that would otherwise terminate the computation, the server executes through the irrelevant computation and proceeds on to process the request (and subsequent requests) successfully. For the other servers (in these servers the memory errors occur in relevant computa-

tions and buffers), failure-oblivious computing converts the attack request (which would otherwise trigger a dangerous, unanticipated execution path) into an anticipated invalid input which the server's standard error-handling logic rejects. The server then proceeds on to read and process subsequent requests acceptably.

One of the reasons that failure-oblivious computing works well for our servers is that they have short error propagation distances — an error in the computation for one request tends to have little or no effect on the computation for subsequent requests. By discarding invalid writes, failure-oblivious computing isolates the effect of any memory errors to data local to the computation for the request that triggered the errors. The result is that the server has short data error propagation distances — the errors do not propagate to data structures required to process subsequent requests. The servers also have short control flow error propagation distances: by preventing addressing exceptions from terminating the computation, failure-oblivious computing enables the server to return to a control flow path that leads it back to read and process the next request. Together, these short data and control flow propagation distances ensure that any effects of the memory error quickly work their way out of the computation, leaving the server ready to successfully process subsequent requests.

1.3 Scope

Our expectation is that failure-oblivious computing will work best with computations, such as servers, that have short error propagation distances. Failure-oblivious computing enables these programs to survive otherwise fatal errors or attacks and to continue on to execute and interact acceptably. Failure-oblivious computing should also be appropriate for multipurpose systems with many components — it can prevent an error in one component from corrupting data in other components and keep the system as a whole operating so that other components can continue to successfully fulfill their purpose in the computation.

Until we develop technology that allows us to track results derived from computations with memory errors, we anticipate that failure-oblivious computing will be less appropriate for programs (such as many numerical computing programs) in which a single error can propagate through to affect much of the computation. We also anticipate that it will be less appropriate for programs in which it is acceptable and convenient to terminate the computation and await external intervention. This situation occurs, for example, during development — the program is typically not producing any useful results and developers with the ability and motivation to find and eliminate any errors are readily available. We therefore see failure-oblivious computing as useful primarily for

deployed programs whose users 1) need the results that the program produces and 2) are unable or unwilling to tolerate failures or to find and fix errors in the program.

1.4 Advantages and Drawbacks

The primary characteristic of failure-oblivious computing as compared with previous approaches is continued execution combined with the elimination of data structure corruption caused by memory errors. The potential benefits include:

- **Availability:** The combination of protection against data structure corruption and continued execution in the face of memory errors can significantly increase the availability of the server. This combination enables the server to continue to provide service to legitimate users even in the face of repeated attacks (or, for that matter, other infrequently-triggered fatal memory errors).
- **Security:** Failure-oblivious computing eliminates the possibility that an attacker can exploit memory errors to corrupt the address space of the server. The result is a more secure system that is immune to buffer-overflow attacks.
- **Minimal Adoption Cost:** The net adoption cost to the developer is to recompile the server using a compiler that generates failure-oblivious code. There is no need to change programming languages, write exception handling code, or modify the software in any way. Failure-oblivious computing can therefore be applied immediately to today's software infrastructure.
- **Reduced Administration Overhead:** One of the most challenging system administration tasks is ensuring that servers are kept up to date with a constant stream of (potentially disruptive) patches and upgrades; this stream is driven, in large part, by the need to eliminate memory-error based security vulnerabilities in otherwise perfectly acceptable servers. Because failure-oblivious computing eliminates this class of errors, it may enable system administrators to safely ignore patches whose purpose is to eliminate security vulnerabilities caused by memory errors. Ideally, administrators would become able to patch their systems primarily to obtain new functionality, not because they need to close security vulnerabilities in programs that are otherwise fully serving the needs of their users.

There are also several potential drawbacks:

- **Unanticipated Execution Paths:** Failure-oblivious computing has the potential to take the program down an execution path that was unanticipated by the programmer, with the prospect of this path producing unacceptable results.¹ This possibility can be especially problematic if errors in the unanticipated path have long propagation distances through the relevant data or when control fails to flow back to an appropriate point in the program. This drawback is, in our view, an unavoidable consequence of *any* mechanism that is intended to increase the resilience of programs in the face of errors — errors occur precisely because the program encountered a situation that the programmer either did not anticipate or did not deem worth handling correctly.
- **The Bystander Effect:** A more abstract issue is the potential for failure-oblivious computing to trigger the *bystander effect* in developers. In a variety of settings that range from manufacturing [25] to personal relationships [40, 24], the mere presence of mechanisms that may detect and compensate for errors has the effect of reducing the effectiveness of the participants in the setting and, in the end, the overall quality of the system as a whole. A potential explanation is that the participants start to rely psychologically on the error recovery mechanisms, which reduces their motivation to eliminate errors in their own work. Deploying failure-oblivious computing into a software development setting may therefore reduce the quality of the software that the developers are able to deliver. One obvious way to combat the bystander effect in this setting is to ban the use of failure-oblivious computing during development. Once again, note that the possibility of triggering the bystander effect is not restricted to failure-oblivious computing — *any* error recovery mechanism has the potential to trigger this effect.

1.5 Contributions

This paper makes the following contributions:

- **Failure-Oblivious Computing:** It introduces the concept of failure-oblivious computing, in which the program discards illegal writes, manufactures values for illegal reads, and continues to execute through memory errors without address space or data structure corruption.

¹We note in passing that this potential is already present in every program — the mere absence of memory errors provides no guarantee that the program is, in fact, operating acceptably.

- **Experience:** It presents our experience using failure-oblivious computing to enhance the security and availability of a range of widely used open-source servers. Our results show that:

- **Standard Compilation:** With the standard unsafe C compiler, the servers are vulnerable to memory errors and attacks that exploit these memory errors.
- **Safe Compilation:** With a C compiler that generates code that exits with an error message when it detects a memory error, the servers exit when presented with an input that triggers a memory error (denying the user access to the services that the server is intended to provide).
- **Failure-Oblivious Compilation:** With our C compiler that generates failure-oblivious code, all of our servers execute successfully through memory errors and attacks to continue to satisfy the needs of their users. Failure-oblivious computing improves both the availability and the security of the servers in our test suite.

- **Explanation:** By relating the properties of servers to the properties of failure-oblivious computing, we explain why failure-oblivious computing may work well for this general class of programs.

2 Example

We next present a simple example that illustrates how failure-oblivious computing operates. Figure 1 presents a (somewhat simplified) version of a procedure from the Mutt mail client discussed in Section 4.6. This procedure takes as input a string encoded in the UTF-8 format and returns as output the same string encoded in modified UTF-7 format. This conversion may increase the size of the string; the problem is that the procedure fails to allocate sufficient space in the return string for the worst-case size increase. Specifically, the procedure assumes a worst-case increase ratio of 2; the actual worst-case ratio is 7/3. When passed (the very rare) inputs with large increase ratios, the procedure attempts to write beyond the end of its output array.

With standard compilers, these writes succeed, corrupt the address space, and the program terminates with a segmentation violation. With safe-C compilers, Mutt exits with a memory error and does not even start the user interface. With our compiler, which generates failure-oblivious code, the program discards all writes beyond the end of the array and the procedure returns with an incompletely translated (truncated) version of the string. Mutt then uses the return value to tell the mail server

```

static char *
utf8_to_utf7 (const char *u8, size_t u8len) {
    char *buf, *p;
    int ch, int n, i, b = 0, k = 0, base64 = 0;

    /* The following line allocates the return
       string. The allocated string is too small;
       instead of u8len*2+1, a safe length would
       be u8len*4+1.
    */
    p = buf = safe_malloc (u8len * 2 + 1);

    while (u8len) {
        unsigned char c = *u8;
        if (c < 0x80) ch = c, n = 0;
        else if (c < 0xc2) goto bail;
        else if (c < 0xe0) ch = c & 0x1f, n = 1;
        else if (c < 0xf0) ch = c & 0x0f, n = 2;
        else if (c < 0xf8) ch = c & 0x07, n = 3;
        else if (c < 0xfc) ch = c & 0x03, n = 4;
        else if (c < 0xfe) ch = c & 0x01, n = 5;
        else goto bail;

        u8++, u8len--;
        if (n > u8len) goto bail;
        for (i = 0; i < n; i++) {
            if ((u8[i] & 0xc0) != 0x80) goto bail;
            ch = (ch << 6) | (u8[i] & 0x3f);
        }
        if (n > 1 && !(ch >> (n*5+1))) goto bail;
        u8 += n, u8len -= n;

        if (ch < 0x20 || ch >= 0x7f) {
            if (!base64) {
                *p++ = '&';
                base64 = 1;
                b = 0;
                k = 10;
            }
            if (ch & ~0xffff) ch = 0xfffe;
            *p++ = B64Chars[b | ch >> k];
            k -= 6;
            for (; k >= 0; k -= 6)
                *p++ = B64Chars[(ch >> k) & 0x3f];
            b = (ch << (-k)) & 0x3f;
            k += 16;
        } else {
            if (base64) {
                if (k > 10) *p++ = B64Chars[b];
                *p++ = '-';
                base64 = 0;
            }
            *p++ = ch;
            if (ch == '&') *p++ = '-';
        }
    }

    if (base64) {
        if (k > 10) *p++ = B64Chars[b];
        *p++ = '-';
    }

    *p++ = '\0';
    safe_realloc ((void **) &buf, p - buf);
    return buf;
}

bail:
    safe_free ((void **) &buf);
    return 0;
}

```

Figure 1: String Encoding Conversion Procedure

which mail folder it wants to open. The mail server responds with an error code indicating that the folder does not exist. Mutt correctly handles this error and continues to execute, enabling the user to process email from other, legitimate, folders.

This example illustrates two key aspects of applying failure-oblivious computing:

- **Subtle Errors:** Real-world programs can contain subtle memory errors that can be very difficult to detect by either testing or code inspection, and these errors can have significant negative consequences for the program and its users.
- **Mostly Correct Programs:** Testing usually ensures that the program is mostly correct and works well except for exceptional operating conditions or inputs. Failure-oblivious computing can therefore be seen as a way to enable the program to proceed past such exceptional situations to return back within its normal operating envelope. And as this example illustrates, failure-oblivious computing can actually facilitate this return by converting unanticipated memory corruption errors into anticipated error cases that the program handles correctly.

3 Implementation

A failure-oblivious compiler generates two kinds of additional code: checking code and continuation code. The checking code detects memory errors and can be the same as in any memory-safe implementation. The continuation code executes when the checking code detects an attempt to perform an illegal access. This code is relatively simple: it discards erroneous writes and manufactures a sequence of values for erroneous reads.

Our implementation uses a checking scheme originally developed by Jones and Kelly [37] and then significantly enhanced by Ruwase and Lam [50]. This checking scheme maintains a table that maps locations to data units (each struct, array, and variable is a data unit) and uses this table to distinguish in bounds and out of bounds pointers.

Our implementation of the write continuation code simply discards the value. Our implementation of the read continuation code redirects the read to a preallocated buffer of values. In principle, any sequence of manufactured values should work. In practice, these values are sometimes used to determine loop conditions. Midnight Commander (see Section 4.5), for example, contains a loop that, for some inputs, searches past the end of a buffer looking for the “/” character. If the sequence of generated values does not include this character, the loop never terminates and Midnight Commander hangs. We therefore generate a sequence that iterates through

all small integers, increasing the chance that, if the values are used to determine loop conditions, the computation will hit upon a value that will exit the loop (and avoid nontermination). Because zero and one are usually the most commonly loaded values in computer programs [59], the sequence is designed to return these values more frequently than other, less common, values.

One potential concern is that failure-oblivious computing may hide errors that would otherwise be detected and eliminated. To help make the errors more apparent, our compiler can optionally augment the generated code to produce a log containing information about the program's attempts to commit memory errors. This log may help administrators to detect and respond appropriately to the presence such errors. Note, however, that hiding errors is one of the primary goals of this research, and that any technique that makes programs more resilient in the face of errors will reduce the negative impact of the errors and therefore the incentive to find and eliminate them.

4 Experience

We implemented a compiler that generates failure-oblivious code, obtained several widely-used open-source servers with known memory errors, and evaluated the impact of failure-oblivious computing on their behavior. Many of these servers are key components of the Linux-based open-source interactive computing environment.

4.1 Methodology

We evaluate the behavior of three different versions of each server: the *Standard* version compiled with a standard C compiler (this version is vulnerable to any memory errors that the server may contain), the *Bounds Check* version compiled with the CRED safe-C compiler [50] (this version terminates the server with an error message at the first memory error), and the *Failure Oblivious* version compiled with our compiler. We evaluate three aspects of each server's behavior:

- **Security and Resilience:** We chose a workload that contains an input that triggers a known memory error in the server; this input typically exploits a security vulnerability as documented by vulnerability-tracking organizations such as Security Focus [13] and SccuriTcam [12]. We observe the behavior of the different versions on this workload; for the Failure Oblivious version we focus on the acceptability of the continued execution after the error.
- **Performance:** We chose a workload that both the Standard and Failure Oblivious versions can execute successfully. We use this workload to measure the *request processing time*, or the time required

for each version to process representative requests. We obtain this time by instrumenting the server to record the time when it starts processing the request and the time when it stops processing the request, then subtracting the start time from the stop time.

- **Stability:** When possible, we deploy the Failure Oblivious version of each server into daily use as part of our normal computational environment. During this deployment we ensure that the workload contains attacks that trigger memory errors in each server. We focus on the long-term acceptability of the continued execution of the Failure Oblivious version of the deployed server.

We note that two of our servers (Pine and Midnight Commander) use out of bounds pointers in pointer inequality comparisons. While this is, strictly speaking, an error, the intention of the programmer is clear. To avoid having these errors cripple the Bounds Check versions of these servers, we (manually) rewrote the code containing the inequality comparisons to eliminate pointer comparisons involving out of bounds pointers.

We ran all the servers on a Dell workstation with two 2.8 GHz Pentium 4 processors, 2 GBytes of RAM, and running Red Hat 8.0 Linux.

4.2 Pine

Pine is a widely used mail user agent (MUA) that is distributed with the Linux operating system [11]. Pine allows users to read mail, fetch mail from an IMAP server, compose and forward mail messages, and perform other email-related tasks. We use Pine 4.44, which is distributed with Red Hat Linux version 8.0. This version of Pine has a memory error associated with a failure to correctly parse certain From fields [10].

4.2.1 The Memory Error

When Pine displays a list of messages, it processes the From field of each message to quote certain characters. This quoting is implemented by transferring the From field into a heap-allocated character buffer for display, inserting a \ character into the buffer before any quoted character. As part of the transfer, the length of the string can increase because of the additional \ characters. The procedure that calculates the maximum possible length of the character buffer fails to correctly account for the potential increase and produces a length that is too short for messages whose From fields contain many quoted characters.

4.2.2 Security and Resilience

The Standard version of Pine writes beyond the end of the buffer, corrupts its heap, and terminates with a segmentation violation. The Bounds Check version detects

the memory error and terminates the computation. With both of these versions, the user is unable to use Pine to read mail because Pine aborts or terminates during initialization as the mail file is loaded and before the user has a chance to interact with the server. The user must manually eliminate the From field from the mail file (using some other mail reader or file editor) before he or she can use Pine to read mail at all.

The Failure Oblivious version discards the out of bounds writes (in effect, truncating the translated From field) and continues to execute through the memory error, enabling the user to process their mail. Because the mail list user interface displays only an initial segment of long From fields, the truncation is not visible to the user. If the user selects the message, a different execution path correctly translates the From field. The displayed message contains the complete From field and the user can read, forward, and otherwise process the message.

4.2.3 Performance

Figure 2 presents the request processing times for the Standard and Failure Oblivious versions of Pine. All times are given in milliseconds. The Read request displays a selected empty message, the Compose request brings up the user interface to compose a message, and the Move request moves an empty message from one folder to another. We performed each request at least twenty times and report the means and standard deviations of the request processing times. All times are given in milliseconds.

Request	Standard	Failure Oblivious	Slowdown
Read	$0.287 \pm 7.1\%$	$1.98 \pm 1.5\%$	6.9
Compose	$0.385 \pm 4.3\%$	$3.11 \pm 2.6\%$	8.1
Move	$1.34 \pm 10.4\%$	$1.80 \pm 11.2\%$	1.34

Figure 2: Request Processing Times for Pine (milliseconds)

Because Pine is an interactive program, its performance is acceptable as long as it feels responsive to its users. Assuming a pause perceptibility threshold of 100 milliseconds for this kind of interactive program [21], it is clear that failure-oblivious computing should not degrade the program's interactive feel. Our subjective experience confirms this expectation: all pause times are imperceptible for all versions.

4.2.4 Stability

During our stability testing period, we used Pine as a default mail reader. Our activities included reading mail, replying to mails, forwarding mails, and managing mail folders. During this time we used Pine to process roughly 25 new mail messages a day (after spam filtering). To test Pine's ability to successfully execute through errors, we

periodically sent an email that triggered the memory error discussed above in Section 4.2.1. We also used the failure-oblivious version of Pine to successfully process a large mail folder containing over 100,000 messages. During this usage period, the Failure Oblivious version executed successfully through all errors to perform all requests flawlessly.

4.3 Apache

The Apache HTTP server is the most widely used web server in the world: a recent survey found that 64% of the web sites on the Internet use Apache [9]. Apache version 2.0.47 contains a (under certain circumstances) remotely exploitable memory error [1].

4.3.1 The Memory Error

Apache can be configured to automatically redirect incoming URLs via a set of URL rewrite rules. Each rewrite rule contains a *match pattern* (a regular expression that may match an incoming URL) and a *replacement pattern*. The match pattern may contain parenthesized *captures*, each of which may match a substring from the incoming URL. The replacement pattern may reference these captures. When an incoming URL matches the match pattern, Apache replaces the URL with the replacement pattern after substituting out any referenced captures with the corresponding captured substrings from the incoming URL. As Apache processes the incoming URL, it uses a (stack-allocated) buffer to hold pairs of offsets that identify the captured substrings in the incoming URL. The buffer contains enough room for ten captures. If there are more, Apache writes the corresponding pairs of offsets beyond the end of the buffer.

4.3.2 Security and Resilience

The Standard version performs the out of bounds writes, corrupts its stack, and terminates with a segmentation violation. The Bounds Check version correctly processes legitimate requests without memory errors until it is presented with a URL that triggers the memory error. At this point the child process serving the connection detects the error and terminates. Apache uses a pool of child processes to serve incoming requests. When one of the child processes terminates, the main Apache process creates a new child process to take its place. This mechanism allows both the Standard and Bounds Check versions of Apache to continue to service requests even when repeatedly presented with inputs that cause the child processes to terminate because of memory errors.

The Failure Oblivious version discards the out of bounds writes and continues to execute. It proceeds on to copy the first ten pairs of offsets into another data structure. Apache uses this data structure to apply the rewrite rule and generate the new URL. Because the rewrite

rule uses a single digit to reference each captured substring (these substrings have names \$0 through \$9), it will never attempt to access any discarded substring offset data. The Failure-Oblivious version of Apache therefore processes each input correctly and continues on to successfully process any subsequent requests. Because the memory errors occur in irrelevant data structures and computations, Failure Oblivious computing eliminates the memory error without affecting the results of the computation at all.

Because Apache isolates request processing inside a pool of regenerating processes, the Bounds Check version successfully processes subsequent requests. The overhead of killing and restarting child processes, however, makes this version vulnerable to an attack that ties up the server by repeatedly presenting it with requests that trigger the error. To investigate this effect, we used several (local) machines to load the server with requests that trigger the error. We then used another client machine to repeatedly fetch the home page of our research project and measured the request throughput at the client. For this workload, the Failure Oblivious version provides a throughput roughly 5.7 times more than the Bounds Check version provides (the insecure Standard version provides a throughput roughly 4.8 times less than the Failure Oblivious version). We attribute the slowdown for the Bounds Check and Standard versions to process management overhead.

4.3.3 Performance

Figure 5 presents the request processing times for the Standard and Failure Oblivious versions of Apache. The Small request serves an 5KByte page (this is the home page for our research project); the large request serves an 830KByte file used only for this experiment. Both requests were local — they came from the same machine on which Apache was running. We performed each request at least twenty times and report the means and standard deviations of the request processing times. All times are given in milliseconds.

Request	Standard	Failure Oblivious	Slowdown
Small	44.4 ± 1.3%	47.1 ± 2.5%	1.06
Large	48.7 ± 1.8%	50.0 ± 1.3%	1.03

Figure 3: Request Processing Times for Apache (milliseconds)

4.3.4 Stability

For the last nine months we have been using the Failure Oblivious version of Apache to serve our research project's web site at www.flexc.csail.mit.edu; during this time period we measured approximately 400 requests a day from outside our institution. We also generated tens

of thousands of requests from another machine, all of which were served correctly. We anticipate that we will continue to use the Failure Oblivious version to serve this web site for the foreseeable future.

During this time period we periodically presented the web server with requests that triggered the vulnerability discussed above. The Failure Oblivious version executed successfully through all of these attacks to continue to successfully service legitimate requests. We observed no anomalous behavior and received no complaints from the users of the web site.

4.4 Sendmail

Sendmail is the standard mail transfer agent for Linux and other Unix systems [15]. It is typically configured to run as a daemon which creates a new process to service each new mail transfer connection. This process executes a simple command language that allows the remote agent to transfer email messages to the Sendmail server, which may deliver the messages to local users or (if necessary) forward some or all of the messages on to other Sendmail servers. Versions of Sendmail earlier than 8.11.7 and 8.12.9 (8.11 and 8.12 are separate development threads) have a memory error vulnerability which is triggered when a remote attacker sends a carefully crafted email message through the Sendmail daemon [14]. We worked with Sendmail version 8.11.6.

4.4.1 The Memory Error

The memory error occurs when Sendmail parses a mail address. A prescan procedure processes the address one character at a time to transfer characters from the address into a fixed-size stack-allocated buffer. This transfer is coded to use a lookahead character and to treat the \ character specially. It is possible for there to be no lookahead character, in which case the integer variable that holds the lookahead character is set to -1. If this variable is set to -1 or contains a \ character that appears in an odd position (first, third, fifth, ...) in a sequence of contiguous \ characters in the address, the prescan skips the block of code that writes the lookahead character into the buffer (also skipping a check to see if the buffer has enough space to hold the lookahead character). It later writes a \ character into the buffer without a check if the lookahead character was \ and not -1. If the execution platform performs sign extension on character to integer assignments, an attack message containing an appropriately placed alternating sequence of -1 and \ characters in the address can therefore cause the prescan to write arbitrarily many \ characters beyond the end of the buffer.

4.4.2 Security and Resilience

The Standard version of Sendmail performs the out of bounds writes and corrupts its call stack. It is apparently

possible for an attacker to exploit the memory error to cause the Sendmail server to execute arbitrary injected code [14]. The Bounds Check version exits with a memory error during initialization and fails to operate at all. The Failure Oblivious version is not vulnerable to the attack — when sent the attack message, it discards the out of bounds writes (preserving the integrity of the stack) and returns back out of the prescan to continue to parse the email address. The next step is to check if the input mail address is too long. This check fails, throwing Sendmail into an anticipated error case. The standard error processing logic in Sendmail then rejects the address, enabling Sendmail to continue on to successfully process subsequent commands.

4.4.3 Performance

Figure 4 presents the means and standard deviations of the request processing times for the Standard and Failure Oblivious versions of Sendmail. All times are given in milliseconds. The Receive Small request receives a message whose body is 4 bytes long; the Send Small request sends the same message. The Receive Large request receives a message whose body is 4Kbytes long; the Send Large request sends the same message. We performed each test at least twenty times to obtain the numbers in Figure 4.

Request	Standard	Failure Oblivious	Slowdown
Recv Small	15.6 ± 2.9%	60.4 ± 1.5%	3.9
Recv Large	16.8 ± 4.3%	65.1 ± 2.3%	3.9
Send Small	20.3 ± 3.2%	75.0 ± 3.4%	3.7
Send Large	21.5 ± 5.7%	76.9 ± 3.8%	3.6

Figure 4: Request Processing Times for Sendmail (milliseconds)

4.4.4 Stability

We installed the Failure Oblivious version of Sendmail on one of our machines and, over the course of several days, used it to send and receive hundreds of thousands of email messages. During this time we repeatedly sent the attack message through the Sendmail daemon, which continued through the attack to correctly process all subsequent Sendmail commands. All of the messages were correctly delivered with no problems. Our memory error logs indicate that Sendmail generates a steady stream of memory errors during its normal execution. In particular, every time the Sendmail daemon wakes up to check for incoming messages, it generates a memory error. This memory error apparently completely disables the Bounds Check version.

4.5 Midnight Commander

Midnight Commander is an open source file management tool that allows users to browse files and archives, copy files from one folder to another, and delete files [6]. Midnight Commander is vulnerable to a memory-error attack associated with accessing an uninitialized buffer when processing symbolic links in `tgz` archives [5]. We used Midnight Commander version 4.5.55 for our experiments.

4.5.1 The Memory Error

Midnight Commander converts absolute symbolic links in `tgz` files into links relative to the start of the `tgz` file. It uses the `strcat` procedure to build up the name of the relative link in a stack-allocated buffer. Unfortunately, the buffer is never initialized. If there are multiple symbolic links in the directory, the component names from all of the links simply accumulate sequentially in the buffer as Midnight Commander processes the set of links. If the combined length of all of the component names exceeds the length of the buffer, `strcat` writes the component names beyond the end of the buffer.

4.5.2 Security and Resilience

The Standard version performs the writes, corrupts its stack, and terminates with a segmentation violation. The Bounds Check version detects the out of bounds access and terminates. The Failure Oblivious version discards the out of bounds writes, enabling Midnight Commander to continue and attempt to look up the data for the referenced file. This lookup always fails (apparently even for the first symbolic link, when the name in the buffer is correct). This is an anticipated case in the Midnight Commander code, which treats the symbolic link as a dangling link and displays it as such to the user. Midnight Commander then continues on to successfully process any subsequent user commands.

4.5.3 Performance

Figure 5 presents the request processing times for the Standard and Failure Oblivious versions of Midnight Commander. The Copy request copies a 31Mbyte directory structure, the Move request moves a directory of the same size, the Mkdir request makes a new directory, and the Delete request deletes a 3.2 Mbyte file. We performed each request at least twenty times and report the means and standard deviations of the request processing times. All times are given in milliseconds.

As these numbers indicate, the Failure Oblivious version is not dramatically slower than the Standard version. Moreover, because Midnight Commander is an interactive program, its performance is acceptable as long as it feels responsive to its users, and these performance results make it clear that the application of failure-

Request	Standard	Failure Oblivious	Slowdown
Copy	$377 \pm 0.7\%$	$535 \pm 2.0\%$	1.4
Move	$0.30 \pm 2.4\%$	$0.406 \pm 1.8\%$	1.4
MkDir	$0.69 \pm 7.0\%$	$1.27 \pm 6.6\%$	1.8
Delete	$2.54 \pm 11.3\%$	$2.72 \pm 11.1\%$	1.1

Figure 5: Request Processing Times for Midnight Commander (milliseconds)

oblivious computing to this program should not degrade its interactive feel. Our subjective experience confirms this expectation: all pause times are imperceptible for both the Standard and Failure Oblivious versions.

4.5.4 Stability

One of the authors uses Midnight Commander on a daily basis as his standard file manipulation tool. During the stability testing period, he used the Failure Oblivious version of Midnight Commander to manage his files. Periodically during the sessions he attempted to open the problematic archive (causing the program to execute through the resulting memory error), then went back to using the Midnight Commander to accomplish his work. Midnight Commander performed without a problem during this time.

The error log shows that Midnight Commander has a memory error that is triggered whenever a blank line occurs in its configuration file. We verified that this error completely disabled the Bounds Check version until we removed the blank lines. The Failure Oblivious version, on the other hand, executed successfully through all memory errors to perform flawlessly for all requests.

4.6 Mutt

Mutt is a customizable, text-based mail user agent that is widely used in the Unix system administration community [8]. It is descended from ELM [4] and supports a variety of features including email threading and correct NFS mail spool locking. We used Mutt version 1.4. As described at [7] and discussed in Section 2, this version is vulnerable to an attack that exploits a memory error in the conversion from UTF-8 to UTF-7 string formats.

4.6.1 The Memory Error

When Mutt opens a mailbox with an IMAP address, it converts the mail folder name from UTF-8 to UTF-7 character encoding. Mutt allocates (in the heap) a temporary character buffer to hold the UTF-7 encoded name. Because UTF-8 to UTF-7 conversion can increase the length of the name, Mutt allocates a buffer twice as long as the UTF-8 name to hold the converted UTF-7 name. However, this buffer is not, in general, long enough — the conversion can increase the length of the UTF-8 name by as much as a factor of 7/3 and not just a factor

of 2. When presented with an appropriately constructed UTF-8 folder name, Mutt writes the converted name beyond the end of the UTF-7 buffer.

4.6.2 Security and Resilience

The Standard version performs the writes, corrupts its heap, and terminates with a segmentation violation. The Bounds Check version detects the memory error and terminates before the user interface comes up. The Failure Oblivious version discards the out of bounds writes, effectively truncating the converted name. Note that even though the UTF-7 buffer may contain no null characters, the folder name is effectively null-terminated: reads beyond the end of the buffer will eventually return null. Once Mutt has obtained the converted folder name, the next step is to place a quoted and escaped version of the name into yet another buffer, then pass this name on as part of a command to the IMAP server. The IMAP server returns an error code indicating that the folder does not exist, Mutt's standard error-handling logic handles the returned error code, and Mutt continues on to successfully process any subsequent user commands.

4.6.3 Performance

Figure 6 presents the request processing times for the Standard and Failure Oblivious versions of Mutt. The Read request reads a selected empty message and the Move request moves an empty message from one folder to another. We performed each request at least twenty times and report the means and standard deviations of the request processing times. All times are given in milliseconds.

Request	Standard	Failure Oblivious	Slowdown
Read	$.655 \pm 4.3\%$	$2.31 \pm 4.8\%$	3.6
Move	$6.94 \pm 6.2\%$	$9.78 \pm 6.2\%$	1.4

Figure 6: Request Processing Times for Mutt (milliseconds)

Because Mutt is an interactive program, its performance is acceptable as long as it feels responsive to its users. These performance results make it clear that the application of failure-oblivious computing to this program should not degrade its interactive feel. Our subjective experience confirms this expectation: all pause times are imperceptible for both the Standard and Failure Oblivious versions.

4.6.4 Stability

During the stability testing period we used the Failure Oblivious version of Mutt to process email messages. We configured Mutt to trigger the security vulnerability described above when it loaded. Mutt successfully executed through the resulting memory errors to correctly

execute all of his requests. We were able to read, forward, and compose mail with no problems even after executing through the memory error. We also used Mutt to process (with no problems) a large mail folder containing over 100,000 messages.

4.7 Discussion

Despite the fact that the dynamic bounds checks have, in theory, the potential to substantially degrade the performance, for several of our servers the overhead is relatively small — the execution times of many of the tasks we measured are apparently dominated by activities (such I/O or operating system functionality) outside the program. Because failure-oblivious computing does not affect the efficiency of these activities, the amortized overhead is relatively small. Moreover, several of our servers are interactive, and interactive tasks can tolerate substantial execution time increases as long as the system maintains its interactive feel. Our results show that failure-oblivious computing maintained acceptable interactive response times for all of our interactive tasks, even for tasks with substantial execution time increases.

For servers, a monitor that detects memory errors and reboots the server when it commits such an error might seem to provide an obvious potential alternative to failure-oblivious computing. Apache, for example, implements a regenerating pool of child processes. The net effect is that the Bounds Check version of Apache can terminate child processes at the first memory error without impairing its ability to continue to service new requests. In comparison with the Failure Oblivious version, the only downside is the performance degradation associated with the resulting increase in process management overhead.

The situation is somewhat different for Pine, Mutt, and Midnight Commander. All of these programs initialize with no memory errors on standard workloads. But once the mailbox contains a message that elicits a memory error (Pine), the system is configured to use a mail folder whose name elicits a memory error (Mutt), or the configuration file contains a blank line (Midnight Commander), the Bounds Check versions exit during initialization. In this situation, restarting is of no use because the restarted computations would, once again, simply exit during initialization. Because these errors are triggered only by carefully crafted or unusual inputs, they could easily make it through a fairly rigorous testing process without being detected. These servers illustrate how aggressively terminating computations at the first memory error can leave deployed systems vulnerable to unanticipated inputs that trigger memory errors and persist or recur in the environment.

Because Sendmail has a memory error whenever it wakes up to check for work, the Bounds Check version

is simply unusable with or without restarting. But note that because the memory errors occur on every execution, it should be possible to use the Bounds Check version to find and eliminate them (as well as any other reproducible memory errors that occur during testing). Even with this change, however, terminating and restarting Sendmail might prove to be problematic — the Sendmail monitor would somehow have to avoid repeatedly presenting Sendmail with messages that triggered a memory error. In contrast, the Failure Oblivious version of Sendmail correctly executed through memory errors to correctly process subsequent messages and the Failure Oblivious version of Pine correctly processed mail messages with headers that elicited memory errors.

5 Related Work

We first note that failure-oblivious computing is an instance of acceptability-oriented computing [47]. Acceptability-oriented computing replaces the concept of program correctness with a set of *acceptability properties* that must hold for the execution of the program to remain acceptable. The programmer then builds and deploys *acceptability enforcement mechanisms* whose actions ensure that these acceptability properties do, in fact, hold. In the case of failure-oblivious computing, the acceptability properties are the absence of memory errors and continued execution; the acceptability enforcement mechanism discards invalid writes and returns manufactured values for invalid reads.

Memory errors, failures, and failure recovery have been core concerns in the field of computer systems since its inception. We discuss related work in these areas.

5.1 Variants and Extensions

We have implemented several variants and extensions of our basic failure-oblivious compiler. These include a compiler that implements *boundless memory blocks* — instead of discarding invalid writes, the generated code stores the values in a hash table indexed under the data unit identifier and offset [48]. Corresponding invalid reads return the appropriate stored values. This variant eliminates size calculation errors — if the program logic is otherwise acceptable, the program will execute acceptably. Another variant redirects out of bounds accesses back into the accessed data unit at an appropriate offset. This strategy may help related sets of out of bounds reads return consistent values from properly initialized data units. Our experience indicates that our set of servers works acceptably with both of these variants.

5.2 Transactional Function Termination

Researchers have also developed a technique to protect servers against buffer-overflow attacks by dynamically detecting buffer overflows, then immediately terminating

the enclosing function and continuing on to execute the code immediately following the corresponding function call [52]. The results indicate that, in many cases, the program can continue on to execute acceptably after the premature function termination. This experience is consistent with our experience that servers can continue to execute successfully through memory errors if they simply discard out of bounds writes and manufacture values for out of bounds reads.

5.3 Safe-C Compilers

Our work builds directly on previous research into memory-safe C implementations [17, 58, 45, 36, 50, 37]. Building on Ruwase and Lam's implementation enabled us to apply failure-oblivious computing directly to legacy programs without modification (some implementations also have this property [58]); some other implementations may require source code changes [22, 38].

It is also feasible to apply failure-oblivious computing to safe languages such as Java or ML by simply replacing the generated code that throws an exception in response to a memory error. As for safe-C implementations, the new code would simply discard illegal writes and return manufactured values for illegal reads.

5.4 Static Analysis

It is also possible to attack the memory error problem directly at its source: a combination of static analysis and program annotations should, in principle, enable programmers to deliver programs that are completely free of memory errors [28, 27, 57, 49]. All of these techniques share the same advantage (a static guarantee that the program will not exhibit a specific kind of memory error) and drawbacks (the need for programmer annotations or the possibility of conservatively rejecting safe programs). Even if the analysis is not able to verify that the entire program is free of memory errors, it may be able to statically recognize some accesses that will never cause a memory error, remove the dynamic checks for those accesses, and thereby reduce any dynamic checking overhead [32, 18, 49].

Researchers have also developed unsound, incomplete analyses that heuristically identify potential errors [54, 19]. The advantage is that such approaches typically require no annotations and scale better to larger programs; the disadvantage is that (because they are unsound) they may miss some genuine memory errors.

5.5 Buffer-Overflow Detection Tools

Researchers have developed techniques that are designed to detect buffer-overflow attacks after they have occurred, then halt the execution of the program before the attack can take effect. StackGuard [23] and StackShield [16] modify the compiler to generate code to detect attacks

that overwrite the return address on the stack; StackShield also performs range checks to detect overwritten function pointers. It is also possible to apply buffer-overflow detection directly to binaries. Purify instruments the binary to detect a range of memory errors, including buffer overruns [34]. Program shepherding uses an efficient binary interpreter to prevent an attacker from executing injected code [39]. A key difference is that failure-oblivious computing prevents the attack from performing the writes that corrupt the address space, which enables the program to continue to execute successfully.

5.6 Rebooting

A traditional and widely used error recovery mechanism is to reboot the system, with repair applied during the reboot if necessary to bring the system back up successfully [30]. Mechanisms such as fast reboots [51] and checkpointing [41, 42] can improve the performance of the basic reboot process.

It is also possible to subdivide (potentially recursively) a system into isolated components, then apply a partial reboot strategy at the granularity of the components. By promoting the construction of the operating system as a collection of small components, microkernel architectures [46, 33, 29] support the application of this approach to operating systems. It is also possible to use mechanisms such as software-based fault isolation [55] or fine-grained hardware memory protection [56] to apply this strategy to selected parts of monolithic operating systems such as kernel extensions. The experimental results show that this approach can eliminate the vast majority of system crashes caused by such extensions [53]. Helper agents are often useful to facilitate the clean termination and reintegration of the restarted component back into the running system (this approach generalizes to support arbitrary recovery actions) [53]. It may also be worthwhile to recursively restart larger and larger subsystems until the system successfully recovers [20].

Failure-oblivious computing differs in that it is designed to keep the system operating through errors instead of restarting. The potential advantages include better availability because of the elimination of down time and the elimination of vulnerabilities to persistent errors. Rebooting, on the other hand, may help ensure that the system stays more closely within the anticipated operating envelope.

5.7 Manual Error Detection and Recovery

Motivated in part by the need to avoid rebooting, researchers have developed more fine-grain error recovery mechanisms. The Lucent 5ESS switch and the IBM MVS operating system, for example, both contain software components that detect and attempt to repair inconsistent data structures [35, 44, 31]. Other techniques

include failure recovery blocks and exception handlers, both of which may contain hand-coded recovery algorithms [43].

To apply these techniques, the programmer must anticipate some aspects of the error and, based on this understanding, develop an appropriate recovery strategy. Failure-oblivious computing, on the other hand, can be applied without programmer intervention to any system and may therefore make the system oblivious to even completely unanticipated errors. Of course, this generality cuts both ways — in particular, failure-oblivious computing may produce less appropriate responses to anticipated errors. We therefore view failure-oblivious computing as largely orthogonal to more application-tailored recovery mechanisms (although failure-oblivious computing may eliminate some of the errors that these mechanisms would otherwise have handled).

Data structure repair [26] occupies a middle ground. Like more traditional error detection and recovery techniques, it requires the programmer to provide some application-specific information (in the case of data structure repair, a data structure consistency specification). But because there is no explicit recovery procedure and because the consistency specification is not tied to specific blocks of code, data structure repair may enable systems to more effectively recover from unanticipated data structure corruption errors.

6 Conclusion

The seemingly inherent brittleness, complexity, and vulnerability (to both errors and attacks) of computer programs can make them frustrating or even dangerous to use. While existing memory-safe languages and memory-safe implementations of unsafe languages may eliminate memory-error vulnerabilities, they can also decrease availability by aggressively throwing exceptions or even terminating the program at the first sign of an error.

Our results show that failure-oblivious computation enhances availability, resilience, and security by continuing to execute through memory errors while ensuring that such errors do not corrupt the address space or data structures of the computation. In many cases failure-oblivious computing can automatically convert unanticipated and dangerous inputs or data into anticipated error cases that the program is designed to handle correctly. The result is that the program survives the unanticipated situation, returns back into its normal operating envelope, and continues to satisfy the needs of its users.

One of the major long-term goals of computer science has been understanding how to build more robust, resilient programs that can flexibly and successfully cope with unanticipated situations. Our research suggests that, remarkably, current systems may already have a substan-

tial capacity for exhibiting this kind of desirable behavior if we only provide a way for them to ignore their errors, protect their data structures from damage, and continue to execute.

Acknowledgements

The authors would like to thank our shepherd David Wagner and the anonymous reviewers for their thoughtful and helpful comments. This research was supported in part by the Singapore-MIT Alliance, DARPA award FA8750-04-2-0254, and NSF grants CCR00-86154, CCR00-63513, CCR00-73513, CCR-0209075, CCR-0341620, and CCR-0325283.

References

- [1] Apache HTTP Server exploit. www.securityfocus.com/bid/8911/discussion/.
- [2] CERT/CC. Advisories 2002. www.cert.org/advisories.
- [3] CNN Report on Code Red. www.cnn.com/2001/TECH/internet/08/08/code.red.ll/.
- [4] ELM. www.instinct.org/elm/.
- [5] Midnight Commander exploit. www.securityfocus.com/bid/8658/discussion/.
- [6] Midnight Commander website. www.ibiblio.org/mc/.
- [7] Mutt exploit. www.securiteam.com/unixfocus/5FP0T0U9FU.html.
- [8] Mutt website. www.mutt.org.
- [9] Netcraft website. http://news.netcraft.com/archives/web_server_survey.html.
- [10] Pine exploit. www.securityfocus.com/bid/6120/discussion.
- [11] Pine website. www.washington.edu/pine/.
- [12] SecuriTeam website. www.securiteam.com.
- [13] Security Focus website. www.securityfocus.com.
- [14] Sendmail exploit. www.securityfocus.com/bid/7230/discussion/.
- [15] Sendmail website. www.sendmail.org.
- [16] Stackshield. www.angelfire.com/sk/stackshield.
- [17] T. Austin, S. Breach, and G. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, June 2004.
- [18] R. Bodik, R. Gupta, and V. Sarkar. Eliminating array bounds checks on demand. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2002.
- [19] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic, programming errors. *Software - Practice and Experience*, 2000.
- [20] G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 110–115, Schloss Elmau, Germany, May 2001.
- [21] S. Card, T. Moran, and A. Newell. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, 1983.
- [22] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, June 2003.
- [23] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Conference*, January 1998.

- [24] J. Darley and B. Latane. Bystander intervention in emergencies: Diffusion of responsibility. *Journal of Personality and Social Psychology*, pages 377–383, Aug. 1968.
- [25] W. E. Deming. *Out of the Crisis*. MIT Press, 2000.
- [26] B. Demsky and M. Rinard. Automatic Detection and Repair of Errors in Data Structures. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2003.
- [27] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. In *Proceedings of the 2003 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, June 2003.
- [28] N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, 2003.
- [29] D. Engler, M. F. Kaashoek, and J. James O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Dec. 1995.
- [30] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
- [31] N. Gupta, L. Jagadeesan, E. Koutsosios, and D. Weiss. Auditdraw: Generating audits the FAST way. In *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering*, 1997.
- [32] R. Gupta. Optimizing array bounds checks using flow analysis. In *ACM Letters on Programming Languages and Systems*, 2(1-4):135-150, March 1993.
- [33] G. Hamilton and P. Kougiouris. The Spring Nucleus: A Microkernel for Objects. In *Proceedings of the 1993 Summer Usenix Conference*, June 1993.
- [34] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, 1992.
- [35] G. Haugk, F. Lax, R. Royer, and J. Williams. The 5ESS(TM) switching system: Maintenance capabilities. *AT&T Technical Journal*, 64(6 part 2):1385–1416, July-August 1985.
- [36] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, June 2002.
- [37] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of Third International Workshop On Automatic Debugging*, May 1997.
- [38] S. C. Kendall. Bcc: run-time checking for C programs. In *USENIX Summer Conference Proceedings*, 1983.
- [39] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of 11th USENIX Security Symposium*, August 2002.
- [40] B. Latane and J. Darley. Group inhibition of bystander intervention in emergencies. *Journal of Personality and Social Psychology*, pages 215–221, Oct. 1968.
- [41] M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, 1988.
- [42] M. Litzkow and M. Solomon. The Evolution of Condor Checkpointing.
- [43] M. R. Lyu. *Software Fault Tolerance*. John Wiley & Sons, 1995.
- [44] S. Mourad and D. Andrews. On the reliability of the IBM MVS/XA operating system. *IEEE Transactions on Software Engineering*, September 1987.
- [45] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages*, 2002.
- [46] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. Jones. Mach: A New Kernel Foundation For UNIX Development. In *Proceedings of the 1986 Summer USENIX Conference*, July 1986.
- [47] M. Rinard. Acceptability-oriented computing. In *2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications Companion (OOPSLA '03 Companion) Onwards! Session*, Oct. 2003.
- [48] M. Rinard, C. Cadar, D. Roy, D. Dumitran, and T. Leu. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *Proceedings of the 20th Annual Computer Security Applications Conference*, Dec. 2004.
- [49] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, June 2000.
- [50] O. Ruwase and M. S. Lam. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, February 2004.
- [51] M. I. Seltzer and C. Small. Self-monitoring and self-adapting operating systems. In *Proceedings of the Sixth workshop on Hot Topics in Operating Systems*, 1997.
- [52] S. Sidiroglou, G. Giovanidis, and A. Keromytis. Using execution transactions to recover from buffer overflow attacks. Technical Report CUCS-031-04, Columbia University Computer Science Department, September 2004.
- [53] M. Swift, B. Bershad, and H. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the Nineteenth ACM Symposium on Operating System Principles*, Dec. 2003.
- [54] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of the Year 2000 Network and Distributed System Security Symposium*, 2000.
- [55] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, Dec. 1994.
- [56] E. Witchel, J. Cates, and K. Asanovic. Mondriaan memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.
- [57] H. Xi and F. Pfenning. Eliminating Array Bound Checking Through Dependent Types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1998.
- [58] S. H. Yong and S. Horwitz. Protecting C Programs from Attacks via Invalid Pointer Dereferences. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, 2003.
- [59] Y. Zhang, J. Yang, and R. Gupta. Frequent value locality and value-centric data cache design. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

Deploying Safe User-Level Network Services with icTCP

Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau

*Computer Sciences Department
University of Wisconsin, Madison
{haryadi, dusseau, remzi}@cs.wisc.edu*

Abstract

We present icTCP, an “information and control” TCP implementation that exposes key pieces of internal TCP state and allows certain TCP variables to be set in a safe fashion. The primary benefit of icTCP is that it enables a variety of TCP extensions to be implemented at user-level while ensuring that extensions are TCP-friendly. We demonstrate the utility of icTCP through a collection of case studies. We show that by exposing information and safe control of the TCP congestion window, we can readily implement user-level versions of TCP Vegas, TCP Nice, and the Congestion Manager; we show how user-level libraries can safely control the duplicate acknowledgment threshold to make TCP more robust to packet reordering or more appropriate for wireless LANs; we also show how the retransmission timeout value can be adjusted dynamically. Finally, we find that converting a stock TCP implementation into icTCP is relatively straightforward; our prototype requires approximately 300 lines of new kernel code.

1 Introduction

Years of networking research have suggested a vast number of modifications to the standard TCP/IP protocol stack [3, 8, 11, 13, 14, 23, 27, 31, 40, 47, 50, 52, 57]. Unfortunately, while some proposals are eventually adopted, many suggested modifications to the TCP stack do not become widely deployed [44].

In this paper, we address the problem of deployment by proposing a small but enabling change to the network stack found in modern operating systems. Specifically, we introduce *icTCP* (pronounced “I See TCP”), a slightly modified in-kernel TCP stack that exports key pieces of state information and provides safe control to user-level libraries. By exposing state and safe control over TCP connections, icTCP enables a broad range of interesting and important network services to be built at user-level.

User-level services built on icTCP are more *deployable* than the same services implemented within the OS TCP stack: new services can be packaged as libraries and easily downloaded by interested parties. This approach is also inherently *flexible*: developers can tailor them to

the exact needs of their applications. Finally, these extensions are *composable*: library services can be used to build more powerful functionality in a lego-like fashion. In general, icTCP facilitates the development of many services that otherwise would have to reside within the OS.

One key advantage of icTCP compared to other approaches for upgrading network protocols [41, 44] is the *simplicity* of implementing the icTCP framework on a new platform. Simplicity is a virtue for two reasons. First, given that icTCP leverages the entire existing TCP stack, it is relatively simple to convert a traditional TCP implementation to icTCP; our Linux-based implementation requires approximately 300 new lines of code. Second, the small amount of code change reduces the chances of introducing new bugs into the protocol; previous TCP modifications often do not have this property [43, 45].

Another advantage of icTCP is the *safe* manner in which it provides new user-level control. Safety is an issue any time users are allowed to modify the behavior of the OS [48]. With icTCP, users are allowed to control only a set of *limited virtual* TCP variables (*e.g.*, *cwnd*, *dupthresh*, and *RTO*). Since users cannot download arbitrary code, OS safety is not a concern. The remaining concern is *network safety*: can applications implement TCP extensions that are not friendly to competing flows [38]? By building on top of the extant TCP Reno stack and by restricting virtual variables to a safe range of values, icTCP ensures that extensions are no more aggressive than TCP Reno and thus are friendly.

In addition to providing simplicity and safeness, a framework such as icTCP must address three additional questions. First, are the overheads of implementing variants of TCP with icTCP reasonable? Our measurements show that services built on icTCP scale well and incur minimal CPU overhead when they use appropriate icTCP waiting mechanisms.

Second, can a wide range of new functionality be implemented using this conservative approach? We demonstrate the utility of icTCP by implementing six extensions of TCP. In the first set of case studies, we focus on modifications that alter the behavior of the transport with

regard to congestion: TCP Vegas [14], TCP Nice [52], and Congestion Manager (CM) [8]. In our second set, we focus on TCP modifications that behave differently in the presence of duplicate acknowledgments: we build a reordering-robust (RR) extension that does not misinterpret packet reordering as packet loss [11, 57] and an extension with efficient fast retransmit (EFR) [50]. In our third set, we explore TCP Eifel [36] which adjusts the retransmit timeout value.

Finally, can these services be developed easily within the framework? We show that the amount of code required to build these extensions as user-level services on icTCP is similar to the original, native implementations.

The rest of this paper is structured as follows. In Section 2 we compare icTCP to related work on extensible network services. In Section 3 we present the design of icTCP and in Section 4 we describe our methodology. In Section 5 we evaluate five important aspects of icTCP: the simplicity of implementing icTCP for a new platform, the network safety ensured of new user-level extensions, the computational overheads, the range of TCP extensions that can be supported, and the complexity of developing those extensions. We conclude in Section 6.

2 Related Work

In this section, we compare icTCP to other approaches that provide networking extensibility.

Upgrading TCP: Four recent projects have proposed frameworks for providing limited extensions for transport protocols; that is, they allow protocols such as TCP to evolve and improve, while still ensuring safety and TCP friendliness. We compare icTCP to these proposals.

Mogul *et al.* [41] propose that applications can “get” and more radically “set” TCP state. In terms of getting TCP state, icTCP is similar to this proposal. The greater philosophical difference arises in how internal TCP state is set. Mogul *et al.* wish to allow arbitrary state setting and suggest that safety can be provided either with a cryptographic signature of previously exported state or by restricting this ability to the super-user. However, icTCP is more conservative, allowing applications to alter parameters only in a restricted fashion. The trade-off is that icTCP can guarantee that new network services are well behaved, but Mogul *et al.*’s approach is likely to enable a broader range of services (*e.g.*, session migration).

The Web100 and Net100 projects [39] are developing a management interface for TCP. Similar to the information component of icTCP, Web100 instruments TCP to export a variety of per-connection statistics; however, Web100 does not propose exporting as detailed information as icTCP (*e.g.*, Web100 does not export timestamps for every message and acknowledgment). The TCP-tuning daemon within Net100 is similar to the control component of icTCP; this daemon observes TCP statistics and responds

by setting TCP parameters [18]. The key difference from icTCP is that Net100 does not propose allowing a complete set of variables to be controlled and does not ensure network safety. Furthermore, Net100 appears suitable only for tuning parameters that do not need to be set frequently; icTCP can frequently adjust in-kernel variables because it provides per-message statistics as well as the ability to block until various in-kernel events occur.

STP [44] also addresses the problem of TCP deployment. STP enables communicating end hosts to remotely upgrade the other’s protocol stack. With STP, the authors show that a broad range of TCP extensions can be deployed. We emphasize two major differences between STP and icTCP. First, STP requires more invasive changes to the kernel to support safe downloading of extension-specific code; support for in-kernel extensibility is fraught with difficulty [48]. In contrast, icTCP makes minimal changes to the kernel. Second, STP requires additional machinery to ensure TCP friendliness; icTCP guarantees friendliness by its very design. Thus, STP is a more powerful framework for TCP extensions, but icTCP can be provided more easily and safely.

Finally, the information component of icTCP is derived from INFOTCP, proposed as part of the infokernel [7]; this previous work showed that INFOTCP enables user-level services to indirectly control the TCP congestion window, *cwnd*. We believe that icTCP improves on INFOTCP in three main ways. First, icTCP exposes information from a more complete set of TCP variables. Second, icTCP allows services to directly set *cwnd* inside of TCP; thus, applications do not need to perform extra buffering nor incur as many sleep/wake events. Finally, icTCP allows TCP variables other than *cwnd* to be controlled. Thus, icTCP not only allows more TCP extensions to be implemented, but is also more efficient and accurate.

User-Level TCP: Researchers have found it useful to move portions of the conventional network stack to user-level [19, 20, 46]. User-level TCP can simplify protocol development in the same way as icTCP. However, a user-level TCP implementation typically struggles with performance, due to extra buffering or context switching or both; further, there is no assurance of network safety.

Application-Specific Networking: A large body of research has investigated how to provide extensibility of network services [22, 28, 37, 51, 53, 54]. These projects allow network protocols to be more specialized to applications than does icTCP, and thus may improve performance more dramatically. However, these approaches tend to require more radical restructuring of the OS or networking stack and do not guarantee TCP friendliness.

Protocol Languages and Architectures: Network languages [1, 35] and structured TCP implementations [10] simplify the development of network protocols. Given the ability to replace or specialize modules, it is

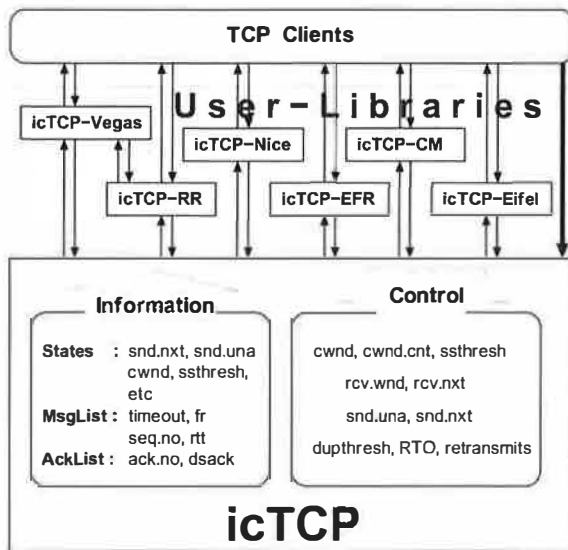


Figure 1: **icTCP Architecture.** The diagram shows the *icTCP* architecture. At the base of the stack is *icTCP*, a slightly modified TCP stack that exports information and limited control. On top of *icTCP*, we have built a number of user-level libraries that implement various pieces of functionality suggested by the literature. The libraries can be composed (where applicable), thus enabling the construction of more powerful services in a plug-and-play fashion. Applications sit at the top of the stack and can choose the libraries that match their needs or directly use the kernel transport.

generally easier to extend existing TCP implementations.

3 icTCP Design

The *icTCP* framework exposes information and provides control over key parameters in the TCP protocol implementation. In this section, we give a high-level overview of how user-level network services are deployed with *icTCP*. We then describe the classes of information and control exported by *icTCP*.

3.1 System Architecture

Figure 1 presents a schematic of the *icTCP* framework. As illustrated, user-level libraries implementing variants of TCP are built on top of *icTCP*. The user-level libraries can be transparently used by applications with standard interfaces. Different TCP connections can use different *icTCP* libraries. The design of *icTCP* is such that only the sending side needs to have *icTCP* deployed; receivers can be running *icTCP* or an unmodified kernel stack.

To simplify the implementation, *icTCP* uses the BSD socket interface, exporting information and providing control with a few new socket options. Although this approach minimized our implementation work, it imposes unnecessary run-time overhead: obtaining state requires a copy from the kernel to user space. Our evaluation shows that user-level network services that naively poll *icTCP* frequently for state information can incur a significant increase in CPU overhead.

To minimize this overhead, *icTCP* provides both a polling and an interrupt-based interface. Given that most TCP variables are updated only when an acknowledgment arrives or at the end of a round (*i.e.*, when one round-trip time has elapsed), applications can receive an interrupt for either condition. In our case studies, the *icTCP* user-level libraries are structured to use two threads; one thread injects packets into the kernel while the other performs sleep/wait and get/set operations.

3.2 Information

The first goal of *icTCP* is to expose information that is traditionally internal to TCP. The challenge is to determine which information should be exposed: if too little information is exposed, it may not be possible to build interesting extensions; if too much information is exposed, then future kernel implementations of TCP may be constrained by an undesirable, expanded interface.

Given that TCP implementations are constrained to adhere to the TCP specification [29], many internal variables are already specified and required. Therefore, *icTCP* explicitly exports all variables that are part of the TCP specification, such as the next sequence number to be sent (*snd.nxt*), the oldest unacknowledged sequence number (*snd.una*), the congestion window (*cwnd*), and the slow start threshold (*ssthresh*). Exposing this information from any TCP implementation should be straightforward.

However, we have found that for more interesting services, access to more information is needed. For example, libraries such as *icTCP-Nice* and *icTCP-RR* must examine information about a particular message. Therefore, *icTCP* exposes “standard” information about each packet. A *message list* provides a history of recent packets, reporting for each packet its sequence number, round-trip time, and whether it is being sent for a time-out or a fast retransmit. An *ack list* provides a history of recent acknowledgments, recording for each packet its acknowledgment number and type (*e.g.*, normal ACK, duplicate ACK, SACK, or DSACK).

Exposing per-packet and per-ack information may not be trivial for those TCP implementations where it does not already exist. Given that TCP Reno does not track the round-trip time of each packet, we add a high resolution timer to *icTCP* to record this information. An additional complexity is that recording new per-message information requires additional memory; therefore, *icTCP* creates these lists only when enabled by user-level services.

3.3 Control

The second goal of *icTCP* is to allow variables that are internal to TCP to be externally set in a safe manner. A new challenge is to determine not only which variables can be modified, but also to what values, while still ensuring that the resulting behavior is TCP-friendly. Our philosophy is

Variable	Description	Safe Range	Example usage
cwnd	Congestion window	$0 \leq v \leq x$	Limit number of sent packets
cwnd.cnt	Linear cwnd increase	$0 \leq v \leq x$	Increase cwnd less aggressively
ssthresh	Slow start threshold	$1 \leq v \leq x$	Move to SS from CA
rcv.wnd	Receive window size	$0 \leq v \leq x$	Reject packet; limit sender
rcv.nxt	Next expected seq num	$x \leq v \leq x + vrcv.wnd$	Reject packet; limit sender
snd.nxt	Next seq num to send	$vsnd.una \leq v \leq x$	Reject ack; enter SS
snd.una	Oldest unacked seq num	$x \leq v \leq vsnd.nxt$	Reject ack; enter FRFR
dupthresh	Duplicate threshold	$1 \leq v \leq vcwnd$	Enter FRFR
RTO	Retransmission timeout	$exp.backoff * (srtt + rttvar) \leq v$	Enter SS
retransmits	Number of consecutive timeouts	$0 \leq v \leq \text{threshold}$	Postpone killing connection

Table 1: Safe Setting of TCP Variables. The table lists the 10 TCP variables which can be set in icTCP. We specify the range each variable can be safely set while ensuring that the result is less aggressive than the baseline TCP implementation. We also give an example usage or some intuition on why it is useful to control this variable. Notation: x refers to TCP's original copy of the variable and v refers to the new virtual copy being set; SS is used for slow start, CA for congestion avoidance, and FRFR for fast retransmit/fast recovery; finally, $srtt$, $rttvar$, and $exp.backoff$ represent smoothed round-trip time, round-trip time variance, and the RTO exponential backoff, respectively.

that icTCP must be conservative: control is only allowed when it is known to not cause aggressive transmission.

The basic idea is that for each variable of interest, icTCP adds a new *limited virtual variable*. Our terminology is as follows: for a TCP variable with the original name *foo*, we introduce a limited virtual variable with the name *vfoo*. However, when the meaning is clear, we simply use the original name. We restrict the range of values that the virtual variable is allowed to cover so that the resulting TCP behavior is friendly; that is, we ensure that the new TCP actions are no more aggressive than those of the original TCP implementation. Given that the acceptable range for a variable is a function of other fluctuating TCP variables, it is not possible to check at call time that the user has specified a valid value and reject invalid settings. Instead, icTCP accepts all settings and coerces the virtual variable into a valid range. For example, the safe range for the virtual congestion window, *vcwnd*, is $0 \leq vcwnd \leq cwnd$. Therefore, if *vcwnd* rises above *cwnd* the value of *cwnd* is used instead.

Converting a variable to a virtual variable within the icTCP stack is not as trivial as it may appear; one cannot simply replace all instances of the original variable with the new virtual one. One must ensure that the virtual value is never used to change the original variable. The simplest case is the statement $cwnd = cwnd + 1$, which clearly cannot be replaced with $cwnd = vcwnd + 1$. More complex cases of control flow currently require careful manual inspection. Therefore, we limit the extent to which the original variable is replaced with the virtual variable.

Given that our foremost goal with icTCP is to ensure that icTCP cannot be used to create aggressive flows, we are conservative in the virtual variables we introduce. Although it would be interesting to allow all TCP variables to be set, the current implementation of icTCP only allows control of ten variables that we are convinced can be safely set from our analysis of the Linux TCP implementation. We do not introduce virtual variables when

the original variable can already be set through other interfaces (e.g., `sysctl` of `tcp_retries1` or `user_mss`) or when they can be approximated in other ways (e.g., we set RTO instead of `srtt`, `mdev`, `rttvar`, or `mrtt`). We do not claim that these ten variables represent the complete collection of settable values, but that they do form a useful set. These ten variables and their safe ranges are summarized in Table 1. We briefly discuss why the specified range of values is safe for each icTCP variable.

The first three variables (i.e., *cwnd*, *cwnd.cnt*, and *ssthresh*) have the property that it is safe to strictly lower their value. In each case, the sender directly transmits less data, because either its congestion window is smaller (i.e., *cwnd* and *cwnd.cnt*) or slow-start is entered instead of congestion avoidance (i.e., *ssthresh*).

The next set of variables determine which packets or acknowledgments are accepted; the constraints on these variables are more complex. On the receiver, a packet is accepted if its sequence number falls inside the receive window (i.e., between *rcv.nxt* and *rcv.nxt* + *rcv.wnd*); thus, increasing *rcv.nxt* or decreasing *rcv.wnd* has the effect of rejecting incoming packets and forces the sender to reduce its sending rate. On the sender, an acknowledgment is processed if its sequence number is between *snd.una* and *snd.nxt*; therefore, increasing *snd.una* or decreasing *snd.nxt* causes the sender to discard acks and again reduce its sending rate. In each case, modifying these values has the effect of dropping additional packets; thus, TCP backs-off appropriately.

The final set of variables (i.e., *dupthresh*, *RTO*, and *retransmits*) control thresholds and timeouts; these variables can be set independently of the original values. For example, both increasing and decreasing *dupthresh* is believed to be safe [57]. Changing these values can increase the amount of traffic, but does not allow the sender to transmit new packets or to increase its congestion window.

Information	LOC	Control	LOC
States	25	cwnd	15
Message List	33	dupthresh	28
Ack List	41	RTO	13
High-resolution RTT	12	ssthresh	19
Wakeup events	50	cwnd.cnt	14
		retransmits	6
		rcv_nxt	20
		rcv_wnd	14
		snd_una	12
		snd_nxt	14
Info Total	161	Control Total	155
icTCP Total			316

Table 2: **Simplicity of Environment.** The table reports the number of C statements (counted with the number of semicolons) needed to implement the current prototype of icTCP within Linux 2.4.

4 Methodology

Our prototype of icTCP is implemented in the Linux 2.4.18 kernel. Our experiments are performed exclusively within the Netbed network emulation environment [56]. A single Netbed machine contains an 850 MHz Pentium 3 CPU with 512 MB of main memory and five Intel EtherExpress Pro 100Mb/s Ethernet ports. The sending endpoints run icTCP, whereas the receivers run stock Linux 2.4.18.

For almost all experiments, a dumbbell topology is used, with one or more senders, two routers interconnected by a (potential) bottleneck link, and one or more receivers. In some experiments, we use a modified Nist-Net [16] on the router nodes to emulate more complex behaviors such as packet reordering. In most experiments, we vary some combination of the bottleneck bandwidth, delay, or maximum queue size through the intermediate router nodes. Experiments are run multiple times (usually 30) and averages are reported; variance is low in those cases where it is not shown.

5 Evaluation

To evaluate whether or not icTCP is a reasonable framework for deploying TCP extensions at user-level, we answer five questions. First, how easily can an existing TCP implementation be converted to provide the information and safe control of icTCP? Second, does icTCP ensure that the resulting network flows are TCP friendly? Third, what are the computation overheads of deploying TCP extensions as user-level processes and how does icTCP scale? Fourth, what types of TCP extensions can be built and deployed with icTCP? Finally, how difficult is it to develop TCP extensions in this way? Note that we spend the bulk of the paper addressing the fourth question concerning the range of extensions that can be implemented and discussing the limitations of our approach.

```
// set internal TCP variables
tcp_setsockopt (option, val) {
    switch (option) {
        case TCP_USE_VCWND:
            use_vwnd = val;
        case TCP_SET_VCWND:
            vcwnd = val;
    }
}

// check if data should be put on the wire
tcp_snd_test () {
    if (use_vwnd)
        min_cwnd = min (vcwnd, cwnd);
    else
        min_cwnd = cwnd;
    // if okay to transmit
    if ((tcp_packets_in_flight < min_cwnd) &&
        /* ... other rules .... */)
        return 1;
    else
        return 0;
}
```

Figure 2: **In-kernel Modification.** Adding *vcwnd* into the TCP stack requires few lines of code. icTCP applications set the virtual variables through the BSD *setsockopt()* interface. Based on the congestion window, *tcp_snd_test* checks if data should be put on the wire. We show that adding a virtual *cwnd* into the decision-making process is simple and straightforward: instead of using *cwnd*, icTCP uses the minimum of *vcwnd* and *cwnd*.

5.1 Simplicity of Environment

We begin by addressing the question of how difficult it is to convert a TCP implementation to icTCP. Our initial version of icTCP has been implemented within Linux 2.4.18. Our experience is that implementing icTCP is fairly straightforward and requires adding few new lines of code. Table 2 shows that we added 316 C statements to TCP to create icTCP. While the number of statements added is not a perfect indicator of complexity, we believe that it does indicate how non-intrusive these modifications are. Figure 2 gives a partial example of how the *vcwnd* variable can be added to the icTCP stack.

5.2 Network Safety

We next investigate whether icTCP flows are TCP friendly. To perform this evaluation, we measure the throughput available to default TCP flows that are competing with icTCP flows. Our measurements show that icTCP is TCP friendly; as desired, the default TCP flows obtain at least as much bandwidth when competing with icTCP as when competing with other default TCP flows. We also show the need for constraining the values into a valid range within icTCP. To illustrate this need, we have created an unconstrained icTCP that allows virtual variables to be set to any value. When default TCP flows compete with unconstrained icTCP flows, the throughput

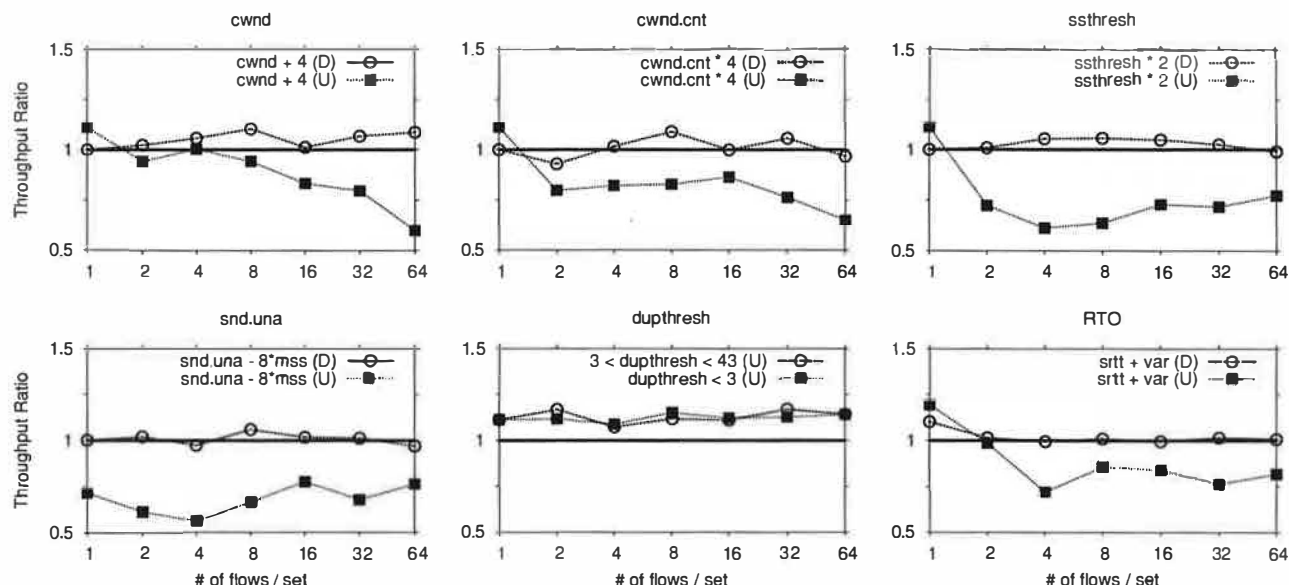


Figure 3: Network Safety of icTCP. Each graph shows two lines: the first line ((D)=Default) uses the default icTCP that enforces parameters to values within their safe range; the second line ((U)=Unconstrained) uses icTCP that allows parameters to be set to any value. The dupthresh graph uses the unconstrained icTCP for both lines. The metric is the ratio of throughput achieved by the default TCP flows when competing with the icTCP flows versus when competing with other default TCP flows. Across the graphs, we vary which icTCP parameters are set; in each case, we set the variable to an unsafe value: *cwnd* to four packets larger, *cwnd.cnt* to four times larger, *ssthresh* to two times larger, *snd.una* to eight packets lower, *dupthresh* to random values below and above three (the default), and *RTO* remaining at an initial *srtt + rtvar* as packets are dropped. The topology used is a dumbbell with four senders and four receivers. For all experiments, except the *RTO* experiments, the bottleneck bandwidth is 100 Mbps with no delay; the *RTO* experiments use a bottleneck bandwidth of 2 Mbps with 15 percent drop rate.

of the default TCP flows is reduced.

Our measurements are shown in Figure 3. Across graphs, we evaluate different icTCP parameters, explicitly setting each parameter to a value outside of its safe range. Along the x-axis of each graph, we increase the number of competing icTCP and TCP flows. Each graph shows two lines: one line has icTCP flows matching our proposal, in which virtual variables are limited to their safe range; the other line has unconstrained icTCP flows. Our metric is the ratio of throughput achieved by the default TCP flows when competing with the icTCP flows versus when competing with other default TCP flows. Thus, if the throughput ratio is around or above one, then the icTCP flows are friendly; if it is below one, then the icTCP flows are unfriendly.

The *cwnd*, *cwnd.cnt*, and *ssthresh* experiments show that these variables must be set within their safe range to ensure friendliness. As expected, icTCP flows that are not allowed to increase their congestion window beyond that of the default TCP remain TCP friendly. Unconstrained icTCP flows that allow larger congestion windows are overly aggressive; as a result, the competing TCP flows obtain less than their fair share of the bandwidth.

We next evaluate the variables that control which acknowledgments or packets are accepted. The behavior for *snd.una* is shown in the fourth graph. The *snd.una* variable represents the highest unacknowledged packet.

When the virtual *snd.una* is set below its safe range of the actual value, then unconstrained icTCP over-estimates the number of bytes acknowledged and increases the congestion window too aggressively. However, when icTCP correctly constrains *snd.una*, the flow remains friendly. The results for the other three variables (i.e., *rcv.wnd*, *rcv.nxt*, and *snd.nxt*) are not shown. In these cases, the icTCP flows remain friendly, as desired, but the unconstrained icTCP flows can fail completely. For example, increasing the *rcv.wnd* variable beyond its safe range can cause the receive buffer to overflow.

The final two graphs explore the *dupthresh* and *RTO* thresholds. We do not experiment with the *retransmits* variable since it is only used to decide when a connection should be terminated. As expected for *dupthresh*, both decreasing and increasing its value from the default of three does not cause unfriendliness; thus, *dupthresh* does not need to be constrained. In the case of *RTO*, the graph shows that if *RTO* is set below $exp.backoff * (srtt + rtvar)$ then the resulting flow is too aggressive.

These graphs represent only a small subset of the experiments we have conducted to investigate TCP friendliness. We have experimented with setting the icTCP variables to random values outside of the safe range and have controlled each of the icTCP parameters in isolation as well as sets of the parameters simultaneously. In all cases, the TCP Reno flows competing with icTCP obtain at least as

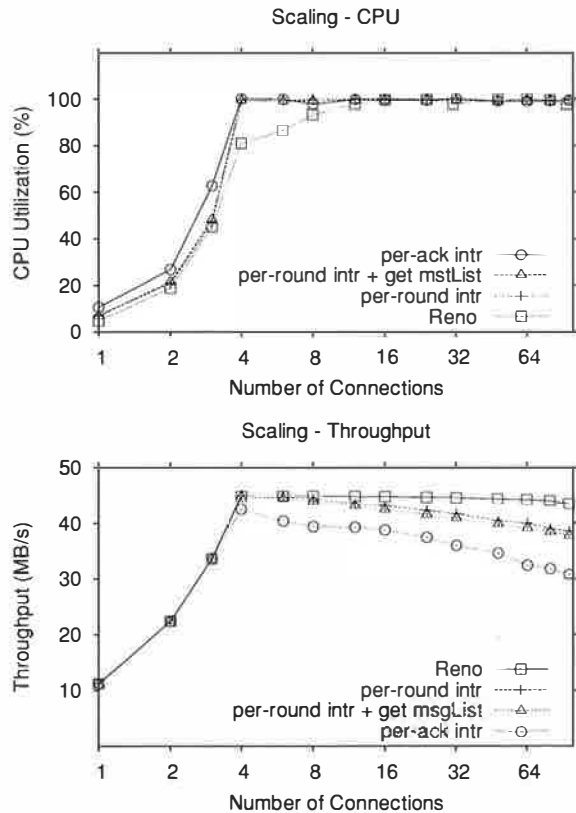


Figure 4: CPU Overhead and Throughput in Scaling icTCP. We connect one sender host to four receiver hosts through different network interfaces. All links are 100 Mbps with no delay links, thus in aggregate the sender host can send data outward at 400 Mbps. Along the x-axis, we increase the number of connections on the sender host. These connections are spread evenly across the four receivers. The first figure compares the overall CPU utilization of Reno, icTCP with per-ack and per-round interrupt. The second figure shows the icTCP throughput degradation when the sender load is high.

much bandwidth as they do when competing with other TCP Reno flows, as desired. In summary, our results empirically demonstrate that icTCP flows require safe variable settings to be TCP friendly. Although these experiments do not prove that icTCP ensures network safety, these measurements combined with our analysis give us confidence that icTCP can be safely deployed.

5.3 CPU Overhead

We evaluate the overhead imposed by the icTCP framework in two ways. First, we explore the scalability of icTCP using synthetic user-level libraries; these experiments explore ways in which a user-level library can reduce CPU overhead by minimizing its interactions with the kernel. Second, we implement TCP Vegas [14] at user-level on top of icTCP; these experiments also allow us to directly compare icTCP to INFOTCP.

5.3.1 Scaling icTCP

We evaluate how icTCP scales as the number of connections is increased on a host. Different user-level extensions built on icTCP are expected to get and set different pieces of TCP information at different rates. The two factors that may determine the amount of overhead are whether the user process requires per-ack or per-round interrupts and whether or not the user process needs the icTCP message list and ack list data structures.

To show the scaling properties of user libraries built on icTCP, we construct three synthetic libraries that mimic the behavior of our later case studies. The first synthetic library uses per-ack interrupts (representing icTCP-EFR and icTCP-Eifel); the second library uses per-round interrupts (icCM); the final library uses per-round interrupts and also gets the message or ack list data structures (icTCP-Vegas, icTCP-Nice, and icTCP-RR).

The two graphs in Figure 4 show how icTCP and TCP Reno scale as the number of flows is increased on a host; the first figure reports CPU utilization and the second figure reports throughput. The first figure shows that icTCP with per-ack and per-round interrupts reaches 100% CPU utilization when there are three and four connections, respectively; the additional CPU overhead of also getting the icTCP message list is negligible. In comparison, TCP Reno reaches roughly 80% utilization with four connections, and then slowly increases to 100% at roughly 16 connections.

The second figure shows that throughput for icTCP starts to degrade when there are four or eight connections, depending upon whether they use per-ack or per-round interrupts, respectively. With 96 flows, the icTCP throughput with per-ack and per-round interrupts is lower than TCP Reno by about 30% and 12%, respectively. Thus, icTCP CPU overhead is noticeable but not prohibitive.

To measure the extent to which a user-level library can accurately implement TCP functionality, we measure the *interrupt miss rate*, defined as how frequently the user misses the interrupt for an ack or the end of a round. In the scaling experiments above with 96 connections, we observed a worst-case miss rate of 1.3% for per-ack interrupts and 0.4% for per-round interrupts. These low miss rates imply that functionality can be placed at the user-level that is responsive to current network conditions.

5.3.2 icTCP-Vegas

To further evaluate icTCP, we implement TCP Vegas congestion avoidance as a user-level library. TCP Vegas reduces latency and increases overall throughput, relative to TCP Reno, by carefully matching the sending rate to the rate at which packets are being drained by the network, thus avoiding packet loss. Specifically, if the sender sees that the measured throughput differs from the expected throughput by more than a fixed threshold, it increases or

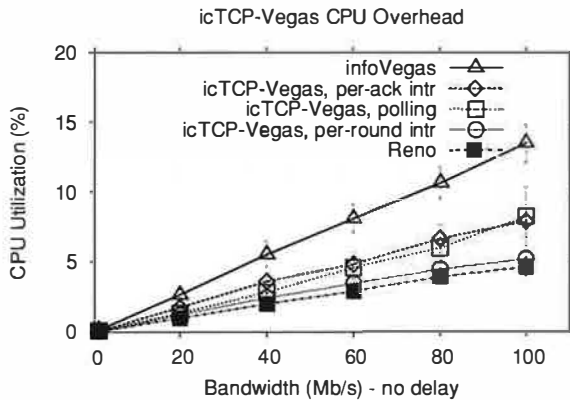


Figure 5: **icTCP-Vegas CPU Overhead.** The figure compares the overall CPU utilization of Reno, INFOVEGAS, and the three versions of icTCP-Vegas. We vary bottleneck-link bandwidth along the x-axis.

decreases its congestion control window, $cwnd$, by one.

Implementation: Our implementation of the Vegas congestion control algorithm, icTCP-Vegas, is structured as follows. The operation of Vegas is placed in a user-level library. This library simply passes all messages directly to icTCP, *i.e.*, no buffering is done at this layer. We implement three different versions that vary the point at which we poll icTCP for new information: every time we send a new packet, every time an acknowledgment is received, or whenever a round ends. After the library gets the relevant TCP state, it calculates its own target congestion window, $vcwnd$. When the value of $vcwnd$ changes, icTCP-Vegas sets this value explicitly inside icTCP.

We note that the implementation of icTCP-Vegas is similar to that of INFOVEGAS, described as part of an infokernel [7]. The primary difference between the two is INFOTCP must manage its own $vcwnd$, as it does not provide control over TCP variables. When INFOVEGAS calculates a value of $vcwnd$ that is less than the actual $cwnd$, INFOVEGAS must buffer its packets and not transfer them to the TCP layer; INFOVEGAS then blocks until an acknowledgment arrives, at which point, it recalculates $vcwnd$ and may send more messages.

Evaluation: We have verified that icTCP-Vegas behaves like the in-kernel implementation of Vegas. Due to space constraints we do not show these results; we instead focus our evaluation on CPU overhead.

Figure 5 shows the total (user and system) CPU utilization as a function of network bandwidth for TCP Reno, the three versions of icTCP-Vegas, and INFOVEGAS. As the available network bandwidth increases, CPU utilization increases for each implementation. The CPU utilization (in particular, system utilization) increases significantly for INFOVEGAS due to its frequent user-kernel crossings. This extra overhead is reduced somewhat for icTCP-Vegas when it polls icTCP on every message send or wakes on the arrival of every acknowledgment, but is

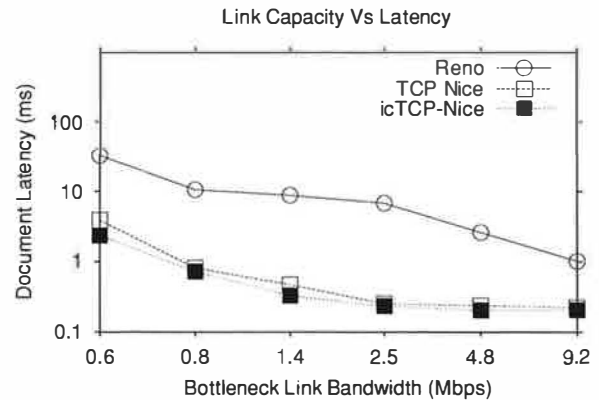


Figure 6: **icTCP-Nice: Link Capacity vs. Latency.** A foreground flow competes with many background flows. Each line corresponds to a different run of the experiment with a protocol for background flows (*i.e.*, icTCP, TCP Nice, Reno, or Vegas). The y-axis shows the average document transfer latency for the foreground traffic. The foreground traffic consists of a 3-minute section of a Squid proxy trace logged at U.C. Berkeley. The background traffic consists of long-running flows. The topology used is a dumbbell with 6 sending nodes and 6 receiving nodes. The foreground flow is alone on one of the sender/receiver pairs while 16 background flows are distributed across the remaining 5 sender/receiver pairs. The bottleneck link bandwidth is varied along the x-axis.

still noticeable. Since getting icTCP information through the *getsockopt* interface incurs significant overhead, icTCP-Vegas can greatly reduce its overhead by getting information less frequently. Because Vegas adjusts $cwnd$ only at the end of a round, icTCP-Vegas can behave accurately while still waking only every round. The optimization results in CPU utilization that is higher by only about 0.5% for icTCP-Vegas than for in-kernel Reno.

5.4 TCP Extensions

Our fourth axis for evaluating icTCP concerns the range of TCP extensions that it allows. Given the importance of this issue, we spend most of the remaining paper on this topic. We address this question by first demonstrating how five more TCP variants can be built on top of icTCP. These case studies are explicitly *not* meant to be exhaustive, but to instead illustrate the flexibility and simplicity of icTCP. We then briefly discuss whether icTCP can be used to implement a wider set of TCP extensions.

5.4.1 icTCP-Nice

In our first case study, we show that TCP Nice [52] can be implemented at user-level with icTCP. This study demonstrates that an algorithm that differs more radically from the base icTCP Reno algorithm can still be implemented. In particular, icTCP-Nice requires access to more of the internal state within icTCP, *i.e.* the complete message list. **Overview:** TCP Nice provides a near zero-cost background transfer; that is, a TCP Nice background flow interferes little with foreground flows and reaps a large fraction of the spare network bandwidth. TCP Nice is

similar to TCP Vegas, with two additional components: multiplicative window reduction in response to increasing round-trip times and the ability to reduce the congestion window below one. We discuss these components in turn.

TCP Nice halves its current congestion window when long round-trip times are measured, unlike Vegas which reduces its window by one and halves its window only when packets are lost. To determine when the window size should be halved, the TCP Nice algorithm monitors round-trip delays, estimates the total queue size at the bottleneck router, and signals congestion when the estimated queue size exceeds a fraction of the estimated maximum queue capacity. Specifically, TCP Nice counts the number of packets for which the delay exceeds $\min RTT + (\max RTT - \min RTT) * t$ (where $t = 0.1$); if the fraction of such delayed packets within a round exceeds f (where $f = 0.5$), then TCP Nice signals congestion and decreases the window multiplicatively.

TCP Nice also allows the window to be less than one; to effect this, when the congestion window is below two, TCP Nice adds a new timer and waits for the appropriate number of RTTs before sending more packets.

Implementation: The implementation of icTCP-Nice is similar to that of icTCP-Vegas, but slightly more complex. First, icTCP-Nice requires information about every packet instead of summary statistics; therefore, icTCP-Nice obtains the full message list containing the sequence number (*seqno*) and round trip time (*usrtt*) of each packet. Second, the implementation of windows less than one is tricky but can also use the *vcwnd* mechanism. In this case, for a window of $1/n$, icTCP-Nice sets *vcwnd* to 1 for a single RTT period, and to 0 for $(n - 1)$ periods.

Evaluation: To demonstrate the effectiveness of the icTCP approach, we replicate several of the experiments from the original TCP Nice paper (*i.e.*, Figures 2, 3, and 4 in [52]).

Our results show that icTCP-Nice performs almost identically to the in-kernel TCP Nice, as desired.

Figure 6 shows the latency of the foreground connections when it competes against 16 background connections and the spare capacity of the network is varied. The results indicate that when icTCP-Nice or TCP Nice are used for background connections, the latency of the foreground connections is often an order of magnitude faster than when TCP Reno is used for background connections. As desired, icTCP-Nice and TCP Nice perform similarly.

The two graphs in Figure 7 show the latency of foreground connections and the throughput of background connections as the number of background connections increases. The graph on the top shows that as more background flows are added, document latency remains essentially constant when either icTCP-Nice or TCP Nice is used for the background flows. The graph on the bottom shows that icTCP-Nice and TCP Nice obtain more

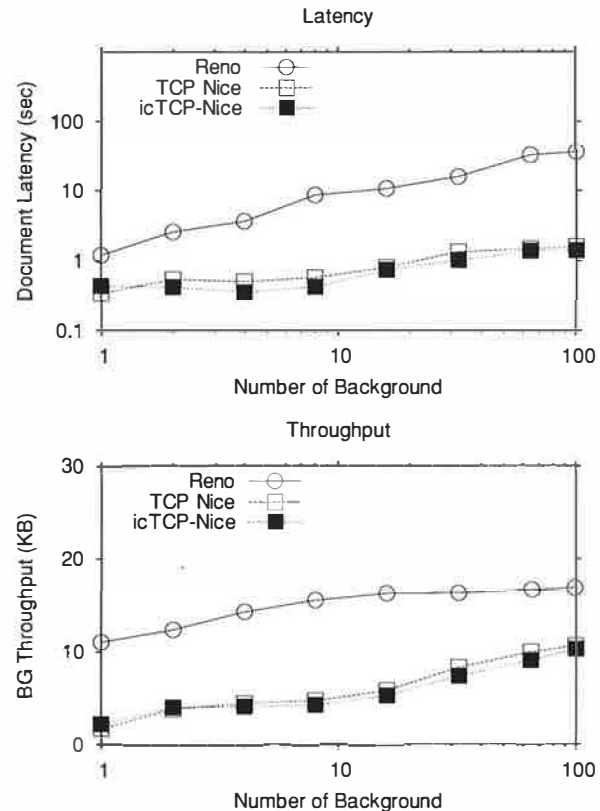


Figure 7: **icTCP-Nice: Impact of Background Flows.** The two graphs correspond to the same experiment; the first graph shows the average document latency for the foreground traffic while the second graph shows the number of bytes the background flows manage to transfer during the 3 minutes period. Each line corresponds to a different protocol for background flows (*i.e.*, TCP Reno, icTCP-Nice, or TCP Nice). The number of background flows is varied along the x-axis. The bottleneck link bandwidth is set to 840 kbps with a 50 ms delay. The experimental setup is identical to Figure 6.

throughput as the number of flows increases. As desired, both icTCP-Nice and TCP Nice achieve similar results.

5.4.2 icCM

We now show that some important components of the Congestion Manager (CM) [8] can be built on icTCP. The main contribution of this study is to show that information can be shared across different icTCP flows and that multiple icTCP flows on the same sender can cooperate.

Overview: The Congestion Manager (CM) architecture [8] is motivated by two types of problematic behavior exhibited by emerging applications. First, applications that employ multiple concurrent flows between sender and receiver have flows that compete with each other for resources, prove overly aggressive, and do not share network information with each other. Second, applications which use UDP-based flows without sound congestion control do not adapt well to changing network conditions.

CM addresses these problems by inserting a module above IP at both the sender and the receiver; this layer

maintains network statistics across flows, orchestrates data transmissions with a new hybrid congestion control algorithm, and obtains feedback from the receiver.

Implementation: The primary difference between icCM and CM is in their location; icCM is built on top of the icTCP layer rather than on top of IP. Because icCM leverages the congestion control algorithm and statistics already present in TCP, icCM is considerably simpler to implement than CM. Furthermore, icCM guarantees that its congestion control algorithm is stable and friendly to existing TCP traffic. However, the icCM approach does have the drawback that non-cooperative applications can bypass icCM and use TCP directly; thus, icCM can only guarantee fairness across the flows for which it is aware.

The icCM architecture running on each sending endpoint has two components: icCM clients associated with each individual flow and an icCM server; there is no component on the receiving endpoint. The icCM server has two roles: to identify macroflows (*i.e.*, flows from this endpoint to the same destination), and to track the aggregate statistics associated with each macroflow. To help identify macroflows, each new client flow registers its process ID and the destination address with the icCM server.

To track statistics, each client flow periodically obtains its own network state from icTCP (*e.g.*, its number of outstanding bytes, `snd.nxt - snd.una`) and shares this state with the icCM server. The icCM server periodically updates its statistics for each macroflow (*e.g.*, sums together the outstanding bytes for each flow in the macroflow). Each client flow can then obtain aggregate statistics for the macroflow for different time intervals.

To implement bandwidth sharing across clients in the same macroflow, each client calculates its own window to limit its number of outstanding bytes. Specifically, each icCM client obtains from the server the number of flows in this macroflow and the total number of outstanding bytes in this flow. From these statistics, the client calculates the number of bytes it can send to obtain its fair share of the bandwidth. If the client is using TCP for transport, then it simply sets `vcwnd` in icTCP to this number. Thus, icCM clients within a macroflow do not compete with one another and instead share the available bandwidth evenly.

Evaluation: We demonstrate the effectiveness of using icTCP to build a congestion manager by replicating one of the experiments performed for CM (*i.e.*, Figure 14 in [8]). In the experiments shown in Figure 8, we place four flows within a macroflow. As shown in the first graph, when four TCP Reno flows are in a macroflow, they do not share the available bandwidth fairly; the performance of the four connections varies between 39 KB/s and 24 KB/s with standard deviation of 5.5 KB/s. In contrast, as shown in the second graph, when four icCM flows are in a macroflow, the connections progress at similar and consistent rates; all four icCM flows achieve throughputs

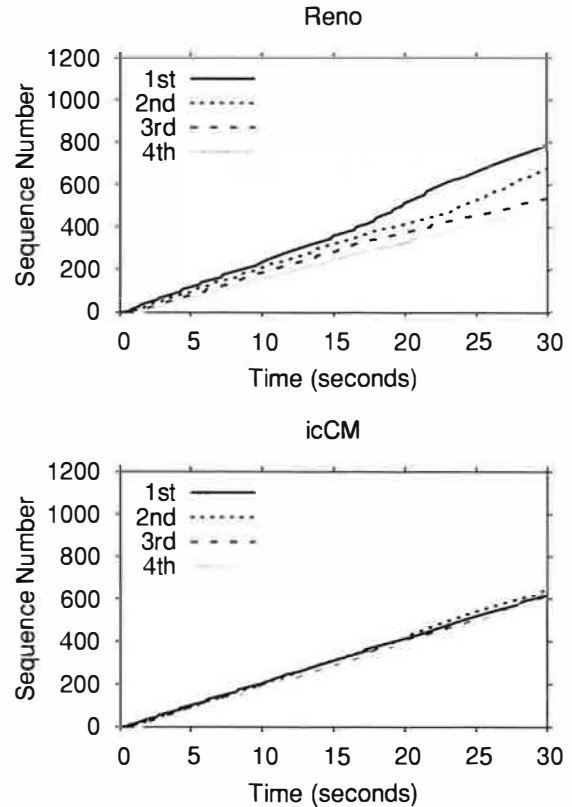


Figure 8: **icCM Fairness.** The two graphs compare the performance of four concurrent transfers from one sender to one receiver, with the bottleneck link set to 1 Mb/s and a 120 ms delay. In the first graph, stock Reno is used; in the second graph, icCM manages the four TCP flows.

of roughly 30 KB/s with a standard deviation of 0.6 KB/s.

5.4.3 icTCP-RR

TCP's fast retransmit optimization is fairly sensitive to the presence of duplicate acknowledgments. Specifically, when TCP detects that three duplicate acks have arrived, it assumes that a loss has occurred, and triggers a retransmission [5, 30]. However, recent research indicates that packet reordering may be more common in the Internet than earlier designers suspected [3, 9, 11, 57]. When frequent reordering occurs, the TCP sender receives a rash of duplicate acks and wrongly concludes that a loss has occurred. As a result, segments are unnecessarily retransmitted (wasting bandwidth) and the congestion window is needlessly reduced (lowering client performance).

Overview: A number of solutions for handling duplicate acknowledgments have been suggested in the literature [11, 57]. At a high level, the algorithms detect the presence of reordering (*e.g.*, by using DSACK) and then increase the duplicate threshold value (`dupthresh`) to avoid triggering fast retransmit. We base our implementation on that of Blanton and Allman's work [11], which limits the maximum value of `dupthresh` to 90% of the window

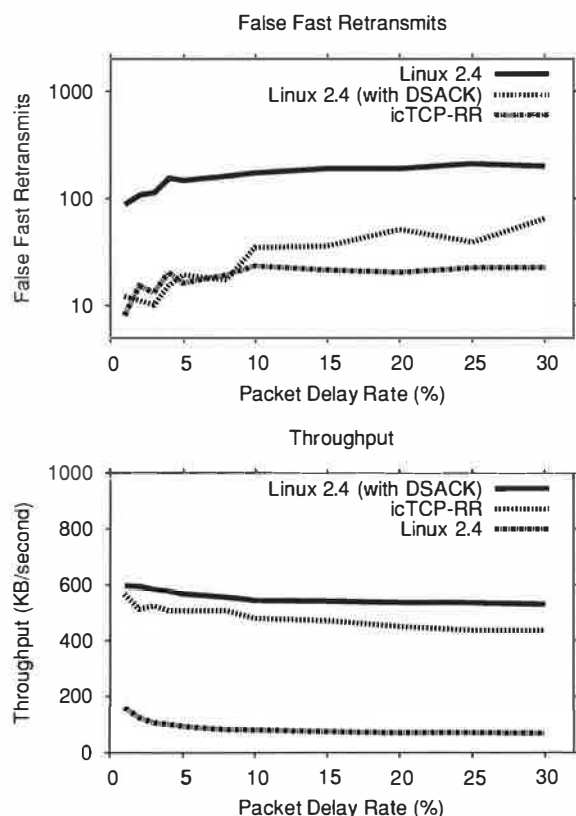


Figure 9: Avoiding False Retransmissions with icTCP-RR. On the top is the number of false retransmissions and on the bottom is throughput, as we vary the fraction of packets that are delayed (and hence reordered) in our modified NistNet router. We compare three different implementations, as described in the text. The experimental setup includes a single sender and receiver; the bottleneck link is set to 5 Mb/s and a 50 ms delay. The NistNet router runs on the first router, introducing a normally distributed packet delay with mean of 25 ms, and standard deviation of 8 ms.

size and, when a timeout occurs, sets *dupthresh* back to its original value of 3.

Implementation: The user-level library implementation, icTCP-RR, is straight-forward. The library keeps a history of acks received; this list is larger than the kernel exported ack list because the kernel may be aggressive in pruning its size, thus losing potentially valuable information. When a DSACK arrives, icTCP places the sequence number of the falsely retransmitted packet into the ack list. The library consults the ack history frequently, looking for these occurrences. If one is found, the library searches through past history to measure the reordering length and sets *dupthresh* accordingly.

Evaluation: Figure 9 shows the effects of packet reordering. We compare three different implementations: stock Linux 2.4 without the DSACK enhancement, Linux 2.4 with DSACK and reordering avoidance built into the kernel, and our user-level icTCP-RR implementation. In the first graph, we show the number of “false” fast retransmis-

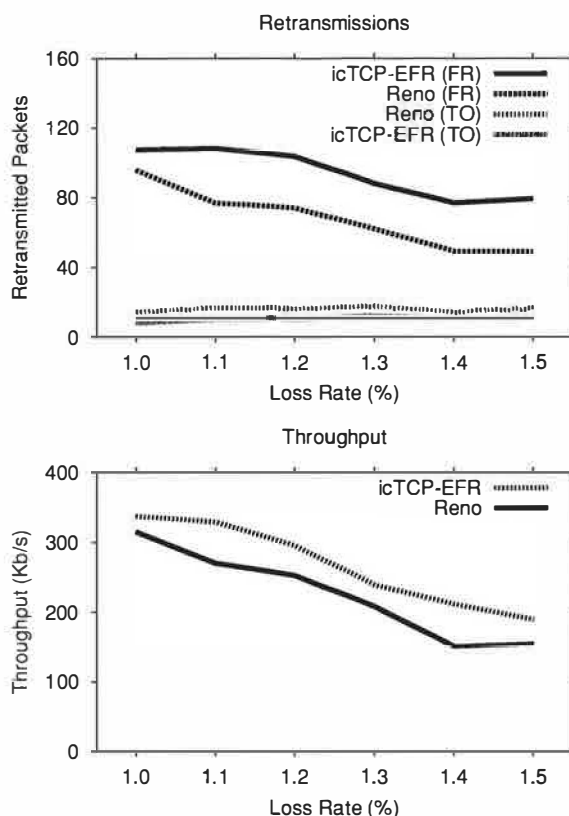


Figure 10: Aggressive Fast Retransmits with icTCP-EFR. On the top is the number of retransmitted packets for both Reno and icTCP-EFR – due to both retransmission timeouts (TO) and fast retransmits (FR) – and on the bottom is the achieved bandwidth. Along the x-axis, we vary the loss rate so as to mimic a wireless LAN. A single sender and single receiver are used, and the bottleneck link is set to 600 Kbps and a 6 ms delay.

sions that occur, where a false retransmission is one that is caused by reordering. One can see that the stock kernel issues many more false retransmits, as it (incorrectly) believes the reordering is actual packet loss. In the second graph, we observe the resulting bandwidth. Here, the DSACK in-kernel and icTCP-RR versions perform much better, essentially ignoring duplicate acks and thus achieving much higher bandwidth.

5.4.4 icTCP-EFR

Our previous case study showed that increasing *dupthresh* can be useful. In contrast, in environments such as wireless LANs, loss is much more common and duplicate acks should be used a strong signal of packet loss, particularly when the window size is small [50]. In this case, the opposite solution is desired; the value of *dupthresh* should be lowered, thus invoking fast retransmit aggressively so as to avoid costly retransmission timeouts.

Overview: We next discuss icTCP-EFR, a user-level library of implementation of EFR (Efficient Fast Retransmit) [50]. The observation underlying EFR is simple: the

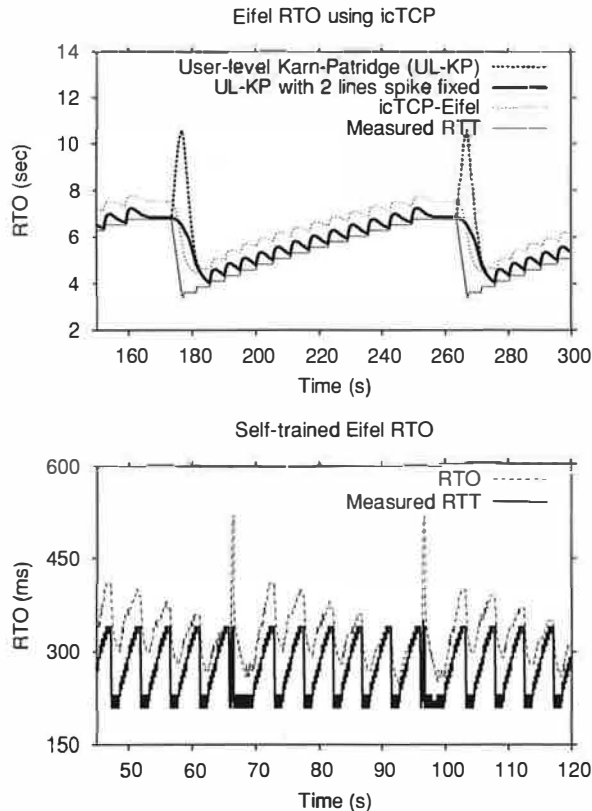


Figure 11: Adjusting RTO with icTCP-Eifel. The graph on the top shows three versions of icTCP-Eifel. For each experiment, the measured round-trip time is identical; however, the calculated RTO differs. The first line shows when the Karn-Partridge RTO algorithm [33] is disabled in the kernel that it can be implemented at user-level with icTCP. In the second experiment, we remove two lines of TCP code that were added to fix the RTO spike; we show that this same fix can be easily provided at user-level. In the third experiment, we implement the full Eifel RTO algorithm at user-level. In these experiments, we emulate a bandwidth of 50 kbps, 1 second delay, and a queue size of 20. The graph on the bottom shows the full adaptive Eifel RTO algorithm with a bandwidth of 1000 kbps, 100 ms delay, and a queue size of 12.

sender should adjust *dupthresh* so as to match the number of duplicate acks it could receive.

Implementation: The icTCP-EFR implementation is also quite straightforward. For simplicity, we only modify *dupthresh* when the window is small; this is where the EFR scheme is most relevant. When the window is small, the library frequently checks the message list for duplicate acks; when it sees one, it computes and sets a new value for *dupthresh*.

Evaluation: Figure 10 shows the behavior of icTCP-EFR versus the in-kernel Reno as a function of loss rate in an emulated wireless network. Because icTCP-EFR interprets duplicate acknowledgments as likely signs of loss, the number of fast retransmits increases (as shown in the graph on top) and more importantly, the number of costly retransmission timeouts is reduced. The graph on the bottom shows that bandwidth increases as a result.

5.4.5 icTCP-Eifel

The retransmission timeout value (RTO) determines how much time must elapse after a packet has been sent until the sender considers it lost and retransmits it. Therefore, the RTO is a prediction of the upper limit of the measured round-trip time (mRTT). Correctly setting RTO can greatly influence performance: an overly aggressive RTO may expire prematurely, forcing unnecessary spurious retransmission; an overly-conservative RTO may cause long idle times before lost packets are retransmitted.

Overview: The Eifel RTO [36] corrects two problems with the traditional Karn-Partridge RTO [33]. First, immediately after mRTT decreases, RTO is incorrectly increased; only after some period of time does the value of RTO decay to the correct value. Second, the “magic numbers” in the RTO calculation assume a low mRTT sampling rate and sender load; if these assumptions are incorrect, RTO incorrectly collapses into mRTT.

Implementation: We have implemented the Eifel RTO algorithm as a user-level library, icTCP-Eifel. This library needs access to three icTCP variables: mRTT, ssthresh, and cwnd; from mRTT, it calculates its own values of srtt (smoothed round-trip) and rttvar (round-trip variance). The icTCP-Eifel library operates as follows: it wakes when an acknowledgment arrives and polls icTCP for the new mRTT; if mRTT has changed, it calculates the new RTO and sets it within icTCP. Thus, this library requires safe control over RTO.

Evaluation: The first graph of Figure 11 shows a progression of three improvements in icTCP-Eifel; these experiments approximately match those in the Eifel RTO paper (i.e., Figure 6 in [36]). In the first implementation, we disable the Karn-Partridge RTO algorithm in the kernel and instead implement it in icTCP-Eifel; as expected, this version incorrectly increases RTO when mRTT decreases. The second implementation corrects this problem with two additional lines of code at user-level; however, RTO eventually collapses into mRTT. Finally, the third version of icTCP-Eifel adjusts RTO so that it is more conservative and avoids spurious retransmissions. The second graph of Figure 11 is similar to Figure 10 in the Eifel paper and shows that we have implemented the full Eifel RTO algorithm at user-level: this algorithm allows RTO to become increasingly aggressive until a spurious timeout occurs, at which point it backs off to a more conservative value.

5.4.6 Summary

From our case studies, we have seen a number of strengths of the icTCP approach. First, icTCP easily enables TCP variants that are less aggressive than Reno to be implemented simply and efficiently at user-level (e.g., TCP Vegas and TCP Nice); thus, there is no need to push such changes into the kernel. Second, icTCP is ideally

suited for tuning parameters whose optimal values depend upon the environment and the workload (*e.g.*, the value of `dupthresh`). Third, `icTCP` is useful for correcting errors in parameter values (*e.g.*, the behavior of `RTO`).

Our case studies have illustrated limitations of `icTCP` as well. From `icCM`, we saw how to assemble a framework that shares information across flows; however, any information that is shared across flows can only be done voluntarily. Furthermore, congestion state learned from previous flows cannot be directly inherited by later flows; this limitation arises from `icTCP`'s reliance upon the in-kernel TCP stack, which cannot be forcibly set to a starting congestion state.

5.4.7 Implementing New Extensions

We evaluate the ability of `icTCP` to implement a wider range of TCP extensions by considering the list discussed for STP [44]. Of the 27 extensions, 9 have already been standardized in Linux 2.4.18 (*e.g.*, `SACK`, `DSACK`, `FAACK`, `TCP` for high performance, `ECN`, `New Reno`, and `SYN cookies`) and 4 have been implemented with `icTCP` (*i.e.*, `RR-TCP`, `Vegas`, `CM`, and `Nice`). We discuss some of the challenges in implementing the remaining 14 extensions. We place these 14 extensions into three categories: those that introduce new algorithms on existing variables, those that modify the packet format, and those that modify the TCP algorithm structure or mechanisms.

Existing Variables: We classify three of the 14 extensions as changing the behavior of existing variables: appropriate byte counting (`ABC`) [2], `TCP Westwood` [55]), and equation-based TCP (`TFRC`) [26]. Other recently proposed TCP extensions that fall into this category include `Fast TCP` [17], `Limited Slow-Start` [25], and `High-Speed TCP` [24].

These extensions are the most natural match with `icTCP` and can be implemented to the extent that they are no more aggressive than `TCP Reno`. For example, equation-based TCP specifies that the congestion window should increase and decrease more gradually than `Reno`; `icTCP-Eqn` allows `cwnd` to increase more gradually, as desired, but forces `cwnd` to decrease at the usual `Reno` rate. We believe that conservative implementations of these extensions are still beneficial. For example, `ABC` implemented on `icTCP` cannot aggressively increase `cwnd` when a receiver delays an ack, but `icTCP-ABC` can still correct for ack division. In the case of `HighSpeed TCP`, the extension cannot be supported in a useful manner because it is strictly more aggressive, specifying that `cwnd` should be decreased by a smaller amount than `TCP Reno` does.

One issue that arises with these extensions is how `icTCP` enforces TCP friendliness: `icTCP` constrains each TCP virtual variable within a safe range, which may be overly conservative. For example, `icTCP` does not allow small increases in TCP's initial congestion window [4],

even though over a long time period these flows are generally considered to be TCP friendly. Alternatively, STP [44] uses a separate module to enforce TCP friendliness; this module monitors the sending rate and verifies that it is below an upper-bound determined by the state of the connection, the mean packet size, the loss event rate, round-trip time, and retransmission timeout. Although `icTCP` could use a similar modular approach, we believe that the equation-based enforcer has an important drawback: non-conforming flows must be terminated, since packets cannot be buffered in a bounded size and then sent at a TCP-friendly rate. Rather than terminate flows, `icTCP` naturally modulates aggressive flows in a manner that is efficient in both space and time.

Packet Format: We classify six of the 14 extensions as changing the format or the contents of packets; for example, extensions that put new bits into the TCP reserved field, such as the `Eifel` algorithm [31] or robust congestion signaling [21]. These extensions cannot be implemented easily with `icTCP` in its current form; therefore, we believe that it is compelling to expand `icTCP` to allow variables in the packet header to be set. However, it may be difficult to ensure that this is done safely.

We can currently approximate this behavior by encapsulating extra information in application data and requiring both the sender and receiver to use an `icTCP`-enabled kernel and an appropriate library; this technique allows extra information to be passed between protocol stacks while remaining transparent to applications. With this technique, we have implemented functionality similar to that of `DCCP` [34]; in our implementation, a user-level library that transmits packets with UDP obtains network information from an `icTCP` flow between the same sender and receiver. We are currently investigating this approach in more detail.

Structure and Mechanism: Approximately five of the 14 extensions modify fundamental aspects of the TCP algorithm: some extensions do not follow the existing TCP states (*e.g.*, `T/TCP` [13] and `limited transmit` [3]) and some define new mechanisms (*e.g.*, the `SCTP` checksum [49]). Given that these extensions deviate substantially from the base `TCP Reno` algorithm, we do not believe that `icTCP` can implement such new behavior.

An approach for addressing this limitation, as well as for modifying packet headers, may be for `icTCP` to provide control underneath the kernel stack with a packet filter [42]. In this way, users could exert control over their packets, perhaps changing the timing, ordering, or altogether suppressing or duplicating some subset of packets as they pass through the filter. Again, such control must be meted out with caution, since ensuring such changes remain TCP friendly is a central challenge.

In summary, `icTCP` is not as powerful as STP [44] and thus can implement a smaller range of TCP extensions.

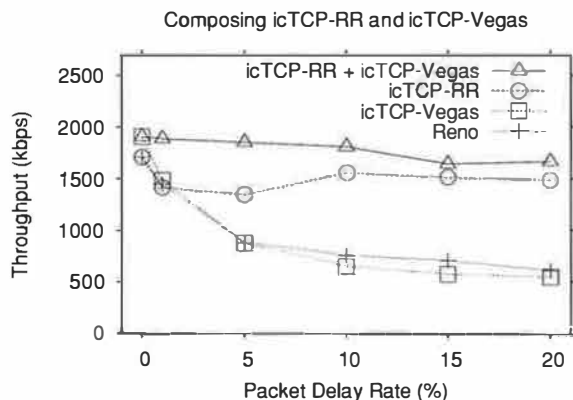


Figure 12: **Composing icTCP-Vegas and icTCP-RR.** The figure shows the strength of composing multiple icTCP libraries in an environment where reordering occurs and the available space in the bottleneck queue is low. When both libraries are used at the same time in this particular environment, the throughput is higher compared to when only one of the libraries is used. The experimental setup includes a single sender and receiver; the bottleneck queue size is set to 5 and the link is set to 2 Mb/s and a 50 ms delay. The NistNet router runs on the first router, introducing a normally distributed packet delay with mean of 25 ms, and standard deviation of 8. On the x-axis we vary the percentage of delayed packets.

However, we believe that the simplicity of providing an icTCP layer on a real system may outweigh this drawback.

5.5 Ease of Development

For our fifth and final question we address the complexity of using the icTCP framework to develop TCP extensions. We answer this question first by showing the ease with which user-level libraries on icTCP can be combined to perform new functionality. We then directly compare the complexity of building TCP extensions at user-level to building them directly in the kernel.

The icTCP framework enables functional composition: given that each user-level library exports the same interface as icTCP, library services can be stacked to build more powerful functionality. In the simplest case, the stacked libraries control disjoint sets of icTCP variables. For example, if the icTCP-Vegas and icTCP-RR libraries are stacked, then the combination controls the values of both *cwnd* and *dupthresh*. Figure 12 shows the advantage of stacking these two libraries: flows running in an environment with both packet reordering and small bottleneck queues exhibit higher throughput with both libraries than with either library alone. Alternatively, the stacked libraries may control overlapping sets of icTCP variables. In this case, each layer further constrains the range of safe values for a virtual variable.

To quantify the complexity of building functionality either on top of icTCP or within the kernel, we count the number of C statements in the implementation (*i.e.*, the number of semicolons), removing those that are used only for printing or debugging. Table 3 shows the number of

Case Study	icTCP	Native
icTCP-Vegas	162	140
icTCP-Nice	191	267
icCM	438	1200*
icTCP-RR	48	26

Table 3: **Ease of Development with icTCP.** The table reports the number of C statements (counted with the number of semicolons) needed to implement the case studies on icTCP compared to a native reference implementation. For the native Vegas implementation, we count the entire patch for Linux 2.2/2.3 [15]. For TCP Nice, we count only statements changing the core transport layer algorithm. For CM, quantifying the number of needed statements is complicated by the fact that the authors provide a complete Linux kernel, with CM modifications distributed throughout; we count only the transport layer. (*) However, this comparison is still not fair given that CM contains more functionality than icCM. For RR, we count the number of lines in Linux 2.4 to calculate the amount of reordering. In-kernel RR uses SACK/DSACK, whereas icTCP-RR traverses the ack list.

C statements required for the four case studies with reference implementations: Vegas, Nice, CM, and RR. Comparing the icTCP user-level libraries to the native implementations, we see that the number of new statements across the two is quite comparable. We conclude that developing services using icTCP is not much more complex than building them natively and has the advantage that debugging and analysis can be performed at user-level.

6 Conclusions

We have presented the design and implementation of icTCP, a slightly modified version of Linux TCP that exposes information and control to applications and user-level libraries above. We have evaluated icTCP across five axes and our findings are as follows.

First, converting a TCP stack to icTCP requires only a small amount of additional code; however, determining precisely where limited virtual parameters should be used in place of the original TCP parameters is a non-trivial exercise. Second, icTCP allows ten internal TCP variables to be safely set by user-level processes; regardless of the values chosen by the user, the resulting flow is TCP friendly. Third, icTCP incurs minimal additional CPU overhead relative to in-kernel implementations as long as icTCP is not polled excessively for new information; to help reduce overhead, icTCP allows processes to block until an acknowledgment arrives or until the end of a round. Fourth, icTCP enables a range of TCP extensions to be implemented at user-level. We have found that icTCP framework is particularly suited for extensions that implement congestion control algorithms that are less aggressive than Reno and for adjusting parameters to better match workload or environment conditions. To support more radical TCP extensions, icTCP will need to be developed further, such as by allowing TCP headers to be safely set or packets and acknowledgments to be reordered or delayed. Fifth, and finally, developing TCP extensions

on top of icTCP is not more complex than implementing them directly in the kernel and are likely easier to debug.

We believe that exposing information and control over other layers in the network stack will be useful as well. For example, given the similarity between TCP and SCTP [6], we believe that SCTP can be extended in a straight-forward manner to icSCTP. An icSCTP framework will allow user-level libraries to again deal with problems such as spurious retransmission [12] as well as implement new functionality for network failure detection and recovery [32].

Our overall conclusion is that icTCP is not quite as powerful as other proposals for extending TCP or other networking protocols [41, 44]. However, the advantage of icTCP is in its simplicity and pragmatism: it is relatively easy to implement icTCP, flows built on icTCP remain TCP friendly, and the computational overheads are reasonable. Thus, we believe that systems with icTCP can, in practice and not just in theory, reap the benefits of user-level TCP extensions.

7 Acknowledgments

The experiments in this paper were performed exclusively in the Netbed network emulation environment from Utah [56]. We are greatly indebted to Robert Ticci, Tim Stack, Leigh Stoller, Kirk Webb, and Jay Lepreau for providing this superb environment for networking research.

We would like to thank Nitin Agrawal, Lakshmi Bairavasundaram, Nathan Burnett, Vijayan Prabhakaran, and Muthian Sivathanu for their helpful discussions and comments on this paper. We would also like to thank Jeffrey Mogul for his excellent shepherding, which has substantially improved the content and presentation of this paper. Finally, we thank the anonymous reviewers for their many helpful suggestions. This work is sponsored by NSF CCR-0092840, CCR-0133456, CCR-0098274, NGS-0103670, ITR-0086044, ITR-0325267, IBM, EMC, and the Wisconsin Alumni Research Foundation.

References

- [1] M. B. Abbott and L. L. Peterson. A Language-based Approach to Protocol Implementation. *IEEE/ACM Transactions on Networking*, 1(1):4–19, Feb. 1993.
- [2] M. Allman. TCP Congestion Control with Appropriate Byte Counting. RFC 3465, Feb. 2002.
- [3] M. Allman, H. Balakrishnan, and S. Floyd. Enhancing TCP's Loss Recovery Using Limited Transmit, Jan. 2001. RFC 3042.
- [4] M. Allman, S. Floyd, and C. Patridge. Increasing TCP's Initial Window. RFC 3390, Internet Engineering Task Force, 2002.
- [5] M. Allman, V. Paxson, and W. R. Stevens. TCP Congestion Control. RFC 2581, Internet Engineering Task Force, Apr. 1999.
- [6] J. Armando L. Caro, J. R. Iyengar, P. D. Amer, S. Ladha, I. Gerard J. Heinz, and K. C. Shah. SCTP: A Proposed Standard for Robust Internet Data Transport. *IEEE Computer*, November 2003.
- [7] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, N. C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J. Nugent, and F. I. Popovici. Transforming Policies into Mechanisms with Infokernel. In *SOSP '03*, 2003.
- [8] H. Balakrishnan, H. S. Rahul, and S. Seshan. An Integrated Congestion Management Architecture for Internet Hosts. In *SIGCOMM '99*, pages 175–187, 1999.
- [9] J. Bellardo and S. Savage. Measuring Packet Reordering. In *Proceedings of the 2002 ACM/USENIX Internet Measurement Workshop*, Marseille, France, Nov. 2002.
- [10] E. Biagioni. A Structured TCP in Standard ML. In *Proceedings of SIGCOMM '94*, pages 36–45, London, United Kingdom, Aug. 1994.
- [11] E. Blanton and M. Allman. On Making TCP More Robust to Packet Reordering. *ACM Computer Communication Review*, 32(1), Jan. 2002.
- [12] E. Blanton and M. Allman. Using TCP DSACKs and SCTP Duplicate TSNs to Detect Spurious Retransmissions. RFC 3708, Internet Engineering Task Force, February 2004.
- [13] R. Braden. T/TCP - TCP Extensions for Transactions. RFC 1644, Internet Engineering Task Force, 1994.
- [14] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *Proceedings of SIGCOMM '94*, pages 24–35, London, United Kingdom, Aug. 1994.
- [15] N. Cardwell and B. Bak. A TCP Vegas Implementation for Linux. <http://flophouse.com/neal/uw/linux-vegas/>.
- [16] M. Carson and D. Santay. NIST Network Emulation Tool. snad.ncsl.nist.gov/nistnet, January 2001.
- [17] D. X. W. Cheng Jin and S. H. Low. FAST TCP: Motivation, Architecture, Algorithms, Performance. In *INFOCOM '04*, 2004.
- [18] T. Dunigan, M. Mathis, and B. Tierney. A TCP Tuning Daemon. In *SC2002*, Nov. 2002.
- [19] A. Edwards and S. Muir. Experiences Implementing a High-Performance TCP in User-space. In *SIGCOMM '95*, pages 196–205, Cambridge, Massachusetts, Aug. 1995.
- [20] D. Ely, S. Savage, and D. Wetherall. Alpine: A User-Level Infrastructure for Network Protocol Development. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS '01)*, pages 171–184, San Francisco, California, Mar. 2001.
- [21] D. Ely, N. Spring, D. Wetherall, and S. Savage. Robust Congestion Signaling. In *ICNP '01*, Nov. 2001.
- [22] M. E. Fiuczynski and B. N. Bershad. An Extensible Protocol Architecture for Application-Specific Networking. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, Jan. 1996.
- [23] S. Floyd. The New Reno Modification to TCP's Fast Recovery Algorithm. RFC 2582, Internet Engineering Task Force, 1999.
- [24] S. Floyd. HighSpeed TCP for Large Congestion Windows. RFC 3649, Internet Engineering Task Force, 2003.
- [25] S. Floyd. Limited Slow-Start for TCP with Large Congestion Windows. RFC 3742, Internet Engineering Task Force, 2004.
- [26] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based Congestion Control for Unicast Applications. In *Proceedings of SIGCOMM '00*, pages 43–56, Stockholm, Sweden, Aug. 2000.
- [27] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An Extension to the Selective Acknowledgment (SACK) Option for TCP. RFC 2883, Internet Engineering Task Force, 2000.
- [28] G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceno, R. Hunt, and T. Pinckney. Fast and Flexible Application-level Networking on Exokernel Systems. *ACM TOCS*, 20(1):49–83, Feb. 2002.

- [29] ISI/USC. Transmission Control Protocol. RFC 793, Sept. 1981.
- [30] V. Jacobson. Congestion avoidance and control. In *Proceedings of SIGCOMM '88*, pages 314–329, Stanford, California, Aug. 1988.
- [31] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323, Internet Engineering Task Force, 1992.
- [32] A. L. C. Jr., J. R. Iyengar, P. D. Amer, and G. J. Heinz. A Two-level Threshold Recovery Mechanism for SCTP. SCI, 2002.
- [33] P. Karn and C. Partridge. Improving Round-Trip Time Estimates in Reliable Transport Protocols. In *Proceedings of SIGCOMM '87*, Aug. 1987.
- [34] E. Kohler, M. Handley, and S. Floyd. Designing DCCP: Congestion Control Without Reliability. www.icir.org/kohler/dccp/dccp-icnp03s.pdf, 2003.
- [35] E. Kohler, M. F. Kaashoek, and D. R. Montgomery. A Readable TCP in the Prolac Protocol Language. In *Proceedings of SIGCOMM '99*, pages 3–13, Cambridge, Massachusetts, Aug. 1999.
- [36] R. Ludwig and K. Sklower. The Eifel Retransmission Timer. *ACM Computer Communications Review*, 30(3), July 2000.
- [37] C. Maeda and B. N. Bershad. Protocol Service Decomposition for High-performance Networking. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 244–255, Asheville, North Carolina, Dec. 1993.
- [38] J. Mahdavi and S. Floyd. TCP-friendly unicast rate-based flow control. end2end-interest mailing list, http://www.psc.edu/networking/papers/tcp_friendly.html, Jan. 1997.
- [39] M. Mathis, J. Heffner, and R. Reddy. Web100: Extended tcp instrumentation. *ACM Computer Communications Review*, 33(3), July 2003.
- [40] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018, Internet Engineering Task Force, 1996.
- [41] J. Mogul, L. Brakmo, D. E. Lowell, D. Subhraveti, and J. Moore. Unveiling the Transport. In *HotNets II*, 2003.
- [42] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The Packet Filter: an Efficient Mechanism for User-level Network Code. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, Austin, Texas, November 1987.
- [43] J. Padhye and S. Floyd. On Inferring TCP Behavior. In *SIGCOMM '01*, pages 287–298, August 2001.
- [44] P. Patel, A. Whitaker, D. Wetherall, J. Lepreau, and T. Stack. Upgrading Transport Protocols using Untrusted Mobile Code. In *SOSP '03*, 2003.
- [45] V. Paxson, M. Allman, S. Dawson, W. Fenner, J. Griner, I. Heavens, K. Lahey, J. Semke, and B. Volz. Known TCP Implementation Problems. RFC 2525, Internet Engineering Task Force, Mar. 1999.
- [46] P. Pradhan, S. Kandula, W. Xu, A. Shaikh, and E. Nahum. Daytona: A user-level tcp stack. <http://nms.lcs.mit.edu/kandula/data/daytona.pdf>, 2002.
- [47] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, Internet Engineering Task Force, 2001.
- [48] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 213–228, Seattle, Washington, Oct. 1996.
- [49] J. Stone, R. Stewart, and D. Otis. Stream control transmission protocol. RFC 3309, Sept. 2002.
- [50] Y. Tamura, Y. Tobe, and H. Tokuda. EFR: A Retransmit Scheme for TCP in Wireless LANs. In *IEEE Conference on Local Area Networks*, pages 2–11, 1998.
- [51] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska. Implementing Network Protocols at User Level. *IEEE/ACM Transactions on Networking*, 1(5):554–565, 1993.
- [52] A. Venkataramani, R. Kokku, and M. Dahlin. Tcp-nice: A mechanism for background transfers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 329–344, Boston, Massachusetts, Dec. 2002.
- [53] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 40–53, Copper Mountain Resort, Colorado, Dec. 1995.
- [54] D. A. Wallach, D. R. Engler, and M. F. Kaashoek. ASHs: Application-specific Handlers for High-performance Messaging. *IEEE/ACM Transactions on Networking*, 5(4):460–474, Aug. 1997.
- [55] R. Wang, M. Valla, M. Sanadidi, and M. Gerla. Adaptive Bandwidth Share Estimation in TCP Westwood. In *IEEE Globecom '02*, 2002.
- [56] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 255–270, Boston, Massachusetts, Dec. 2002.
- [57] M. Zhang, B. Karp, S. Floyd, and L. Peterson. RR-TCP: A Reordering-Robust TCP with DSACK. In *11th International Conference on Network Protocols (ICNP '03)*, June 2003.

ksniffer: Determining the Remote Client Perceived Response Time from Live Packet Streams

David P. Olshefski^{1,2}, Jason Nieh¹ and Erich Nahum²

¹Columbia University and ²IBM T.J Watson Research

olshef@us.ibm.com, nieh@cs.columbia.edu, nahum@us.ibm.com

Abstract

As dependence on the World Wide Web continues to grow, so does the need for businesses to have quantitative measures of the client perceived response times of their Web services. We present *ksniffer*, a kernel-based traffic monitor capable of determining pageview response times as perceived by remote clients, in real-time at gigabit traffic rates. *ksniffer* is based on novel, online mechanisms that take a “look once, then drop” approach to packet analysis to reconstruct TCP connections and learn client pageview activity. These mechanisms are designed to operate accurately with live network traffic even in the presence of packet loss and delay, and can be efficiently implemented in kernel space. This enables *ksniffer* to perform analysis that exceeds the functionality of current traffic analyzers while doing so at high bandwidth rates. *ksniffer* requires only to passively monitor network traffic and can be integrated with systems that perform server management to achieve specified response time goals. Our experimental results demonstrate that *ksniffer* can run on an inexpensive, commodity, Linux-based PC and provide online pageview response time measurements, across a wide range of operating conditions, that are within five percent of the response times measured at the client by detailed instrumentation.

1 Introduction

For many businesses, the World Wide Web is a highly competitive environment. Customers seeking quality online services have choices, and often the characteristic that distinguishes a successful site from the rest is performance. Clients are keenly aware when response time exceeds acceptable thresholds and are not hesitant to simply take their business elsewhere. It is therefore extremely important for businesses to know the response time that their clients are experiencing. This places them in a difficult position: having to obtain accurate client perceived response time metrics in a timely, cost effective manner so that problems can be immediately identified and fixed. For larger Web sites, the requirement of having a scalable solution is key; in addition, the capability to transmit this information to an online cluster management system is also a necessity.

Server farm management systems that allocate resources on-demand to meet specified response time goals are receiving much attention. The ability of a Web hosting center to move CPU cycles, machines, bandwidth and storage from a hosted Web site that is meeting its latency goal to one that is not, is a key requirement for an automated management system. Such allocation decisions must be based on accurate measurements. Over-allocating resources to one hosted Web site results in an overcharge to that customer and a reduction in the available physical resources left to meet the needs of the others. Under-allocation results in poor response time and unsatisfied Web site users. The ability to base these allocation decisions on a measure that is relevant to both the Web site owner and the end user of the Web site is a competitive advantage.

Unfortunately, obtaining an accurate measure of the client perceived response time is non-trivial. Current approaches include active probing from geographically distributed monitors, instrumenting HTML Web pages with JavaScript, offline analysis of packet traces, and instrumenting Web servers to measure application-level performance or per connection performance. All of these approaches fall short, in one area or another, in terms of accuracy, cost, scalability, usefulness of information collected, and real-time availability of measurements.

We have created *ksniffer*, an online server-side traffic monitor that combines passive packet capture with fast online mechanisms to accurately determine client perceived pageview response times on a per pageview basis. *ksniffer* uses a model of TCP retransmission and exponential backoff that accounts for latency due to connection setup overhead and network packet loss. It combines this model with higher level online mechanisms that use access history and HTTP referer information when available to learn relationships among Web objects to correlate connections and Web objects to determine pageview response times.

ksniffer mechanisms take a “look once, then drop” approach to packet analysis, use simple hashing data structures to match Web objects to pageviews, and can be efficiently implemented in kernel space. Furthermore, *ksniffer* only looks at TCP/IP and HTTP protocol header information and does not need to parse any HTTP data

payload. This enables ksniffer to perform higher level Web pageview analysis effectively online in the presence of high data rates; it can monitor traffic at gigabit line speeds while running on an inexpensive, commodity PC. These mechanisms enable ksniffer to provide accurate results across a wide range of operating conditions, including high load, connection drops, and packet loss. In these cases, obtaining accurate performance measures is most crucial because Web server and network resources may be overloaded.

ksniffer has several advantages over other approaches. First, ksniffer does not require any modifications to Web pages, Web servers, or browsers, making deployment easier and faster. This is particularly important for Web hosting companies responsible for maintaining the infrastructure surrounding a Web site but are often not permitted to modify the customer's server machines or content. Second, ksniffer captures network characteristics such as packet loss and delay, aiding in distinguishing network problems from server problems. Third, ksniffer measures the behavior of every session for every real client who visits the Web site. Therefore, it does not fall prey to biases that arise when sampling from a select, predefined set of client monitoring machines that have better connectivity, and use different Web browser software, than the actual users of the Web site. Fourth, ksniffer can obtain metrics for any Web content, not just HTML. Fifth, ksniffer performs online analysis of high bandwidth, live packet traffic instead of offline analysis of traces stored on disk, bypassing the need to manage large amounts of disk storage to store packet traces. More importantly, ksniffer can provide performance measurements to Web servers in real-time, enabling them to respond immediately to performance problems through diagnosis and resource management.

This paper presents the design and implementation of ksniffer. Section 2 presents an overview of the ksniffer architecture. Section 3 describes the ksniffer algorithms for reconstructing TCP connections and pageview activities. Section 4 discusses how ksniffer handles less ideal operating conditions, such as packet loss and server overload. Section 5 presents experimental results quantifying the accuracy and scalability of ksniffer under various operating conditions. We measure the accuracy of ksniffer against measurements obtained at the client and compare the scalability of ksniffer against user-space packet analysis systems. Section 6 discusses related work. Finally, we present some concluding remarks and directions for future work.

2 Overview of ksniffer Architecture

ksniffer is motivated by the desire to have a fast, scalable, flexible, inexpensive traffic monitor that can be used both in production environments for observing Web servers,

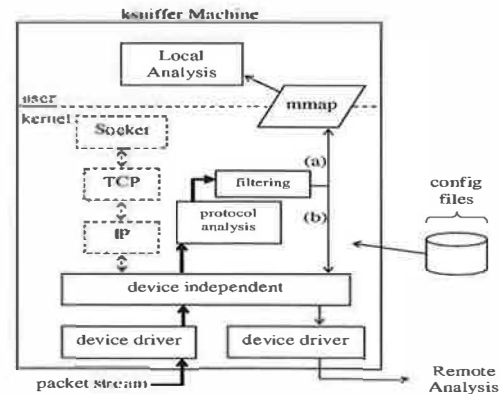


Figure 1: ksniffer architecture.

as well as a platform for research into traffic analysis. Figure 1 depicts the ksniffer architecture.

ksniffer is designed to be implemented as a set of dynamically loadable kernel modules that reside above the network device independent layer in the operating system. Its device independence makes it easy to deploy on any inexpensive, commodity PC without special NIC hardware or device driver modifications. ksniffer appears to the kernel simply as another network protocol layer within the stack and is treated no different than TCP/IP, which is shown for comparison in Figure 1. ksniffer monitors bidirectional traffic and looks at each packet once, extracts any TCP/IP or HTTP header information that is present, then discards the packet. The in-kernel implementation exploits several performance advantages such as zero-copy buffer management, eliminated system calls, and reduced context switches [16, 17]. ksniffer does not produce packet trace log files, but can read configuration parameters and write debugging information to disk from kernel space.

This design gives ksniffer a three to four fold improvement in performance over user space systems that copy every packet to user space. Each packet could potentially impact the response time measurement, yet ksniffer only examines a small percentage of the bytes within each packet (TCP/IP fields and the HTTP headers, if present). By executing in kernel space, ksniffer avoids transferring large amounts of irrelevant bytes to user space, saving CPU cycles and memory bandwidth.

ksniffer provides a low overhead shared memory interface (similar to MAGNET [13]) to export results (*not packets*) to user space. This allows more sophisticated analysis that is less performance critical to be done in user-level programs without additional system call overhead. ksniffer also provides the ability to transmit results directly to a remote machine for processing. Filtering within ksniffer is performed on the results, not on the incoming packet stream. This differentiates ksniffer from traditional monitors that exclude certain TCP flows from analysis, which affects aggregate metrics for

the Web site. A detailed discussion of how ksniffer facilitates other user-level and remote analysis is beyond the scope of this paper. The focus of this paper is on the protocol analysis portion of ksniffer shown in Figure 1, which contains the functionality for determining pageview response times. For simplicity, we assume a single Web server in our discussion, but the same ksniffer monitoring approach also applies to a Web site supported by multiple Web servers.

3 ksniffer Pageview Response Time

To determine the client perceived response time for a Web page, ksniffer measures the time from when the client sends a packet corresponding to the start of the transaction until the client receives the packet corresponding to the end of the transaction. How a packet may indicate the start or end of a transaction depends upon several factors. To show how this is done, we first briefly describe some basic entities tracked by ksniffer, then describe how ksniffer determines response time based on an anatomical view of the client/server behavior that occurs when a Web page is downloaded.

ksniffer keeps track of four entities to maintain the information it needs to measure response time: clients, pageviews, HTTP objects, and TCP connections. ksniffer tracks each of these entities using the corresponding data objects shown in Figure 2. Clients are uniquely identified by their IP address. A pageview consists of a container page and a set of embedded HTTP objects. For example, a typical Web page consists of an HTML file as the container page and a set of embedded images which are the embedded HTTP objects. Pageviews are identified by the URL of the associated container page and Web objects are identified by their URL. A flow represents a TCP connection, and is uniquely identified by the four tuple consisting of source and destination IP address and port numbers.

It is the associations between instances of these objects which enables ksniffer to reconstruct the activity at the Web site. To efficiently manage these associations, ksniffer maintains sets of hash tables to perform fast lookup and correlation between the four types of objects. Separate hash tables are used for finding clients and flows, indexed by hash functions on the IP address and four-tuple, respectively. Each client object contains a pageview hash table indexed by a hash function over the container page URL. Flows contain a FIFO request queue of Web objects that have been requested but not completed, and a FIFO finish queue of Web objects that have been completed.

Suppose a remote client, C_j , requests a Web page. We decompose the resulting client/server behavior into four parts: TCP connection setup, HTTP request, HTTP response, and embedded object processing. We use the fol-

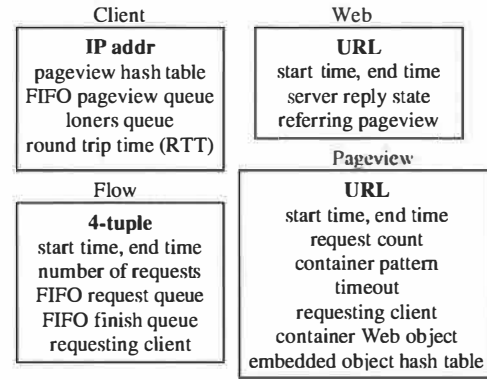


Figure 2: Objects used by ksniffer for tracking.

lowing notation in our discussion. Let C_j be the j^{th} remote client and F_i^j be the i^{th} TCP connection associated with remote client C_j . Let pv_i^j be the i^{th} pageview associated with remote client C_j , and $w_k^{j,i}$ be the k^{th} Web object requested on F_i^j . Let t_i be the i^{th} moment in time, d represent an insignificant amount of processing time, either at the client or the server, p represent the Web server processing time of an HTTP request, and RTT be the round trip time between the client and the server.

3.1 TCP Connection Setup

If the client, C_j , is not currently connected to the Web server, the pageview transaction begins with making a connection. Connection establishment is performed using the well known TCP three-way handshake, as shown in Figure 3. The start of the pageview transaction corresponds to the SYN J packet transmitted by the client at time t_0 . However, ksniffer is located on the server-side of the network, where a dotted line is used in Figure 3 to represent the point at which ksniffer captures the packet stream. ksniffer does not capture SYN J until time $t_0 + .5RTT$, after the packet takes $1/2$ RTT to traverse the network. This is assuming ksniffer and the Web server are located close enough together that they see packets at essentially the same time.

If this is the first connection from C_j , ksniffer will create a flow object F_1^j and insert it in the flow hash table. At this moment, ksniffer does not know the value for RTT since only the SYN J packet has been captured, so it cannot immediately determine time t_0 . Instead, it sets the start time for F_1^j equal to $t_0 + .5RTT$. ksniffer then waits for further activity on the connection. At $t_0 + 1.5RTT + 2d$, ksniffer and the Web server receive the ACK K+1 packet, establishing the TCP connection between client and server. ksniffer can now determine the RTT as the difference between the SYN-ACK from the server (the SYN K, ACK J+1 packet) and the resulting ACK from the client during connection establishment (the ACK K+1 packet). ksniffer then updates F_1^j 's start time by subtracting $1/2$ RTT from its value to obtain t_0 .

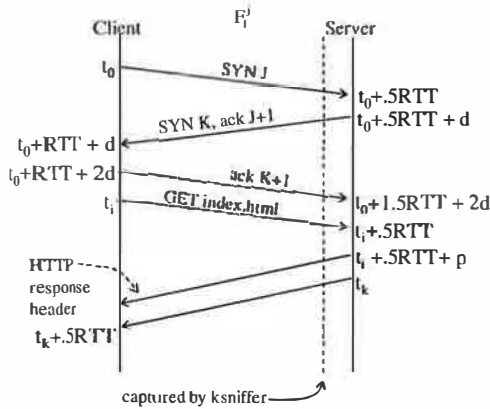


Figure 3: HTTP request/reply.

At time $t_0 + 1.5RTT + 2d$, for the first connection from C_j , ksniffer creates a client object C_j , saves the RTT value, and inserts the object into the client hash table. For each subsequent connection from C_j , a new flow object F_i^j will be created and linked to the existing client object, C_j . The RTT for each new flow will be computed, and C_j 's RTT will be updated based on an exponentially weighted moving average of the RTT s of its flows in the same manner as TCP [28]. The updated RTT is then used to determine the actual start time for each flow, t_0 .

3.2 HTTP Request

Once connected to the server, the remote client transmits an HTTP request for the container page and waits for the response. If this is not the first request over the connection, then this HTTP request indicates the beginning of the pageview transaction. Figure 3 depicts the first request over a connection. At time t_i , the client transmits the HTTP GET request onto the network, and after taking $1/2 RTT$ to traverse the network, the server receives the request at $t_i + .5RTT$.

ksniffer captures and parses the packet containing the HTTP GET request, splitting the request into all its constituent components and identifying the URL requested. Since this is the first HTTP request over connection F_1^j , it incurs the connection setup overhead. In this case, a Web object is created, $w_1^{j,1}$, to represent the request, and the start time for $w_1^{j,1}$ is set to the start time of F_1^j . In this manner, the connection setup time is attributed to the first HTTP request on each flow. $w_1^{j,1}$ is then inserted into F_1^j 's request queue and F_1^j 's number-of-requests field is set to one. If this was not the first HTTP request over connection F_1^j , but was instead the k^{th} request on F_1^j , a Web object $w_k^{j,1}$ would be created but its start time would be set equal to t_i .

Next, ksniffer creates pv_1^j , the pageview object that will track the pageview, and inserts it into C_j 's pageview hash table. We assume for the moment that $w_1^{j,1}$ is a

container page; embedded objects are discussed in Section 3.5. ksniffer sets pv_1^j 's start time equal to $w_1^{j,1}$'s start time, and sets $w_1^{j,1}$ as the container Web object for pv_1^j . At this point in time, ksniffer has properly determined which pageview is being downloaded, and the correct start time of the transaction.

3.3 HTTP Response

After the Web server receives the HTTP request and takes p amount of time to process it, the server sends a reply back to the client. ksniffer captures the value of p , the server response time, which is often mistakenly cited as the client perceived response time. Server response time can underestimate the client perceived response time by more than an order of magnitude [27]. The first response packet contains the HTTP response header, along with the initial portion of the Web object being retrieved. ksniffer looks at the response headers but never parses the actual Web content returned by the server; HTML parsing would entail too much overhead to be used in an online, high bandwidth environment.

ksniffer obtains F_1^j from the flow hash table and determines the first Web object in F_1^j 's request queue is $w_1^{j,1}$, which was placed onto the queue when the request was captured. An HTTP response header does not specify the URL for which the response is for. Instead, HTTP protocol semantics dictate that, for a given connection, HTTP requests be serviced in the order they are received by the Web server. As a result, F_1^j 's FIFO request queue enables ksniffer to identify each response over a flow with the correct request object.

ksniffer updates $w_1^{j,1}$'s server reply state based on information contained in the response header. In particular, ksniffer uses the *Content-length:* and *Transfer-Encoding:* fields, if present, to determine what will be the sequence number of the last byte of data transmitted by the server for this request.

ksniffer captures each subsequent packet to identify the time of the end of the response. This is usually done by identifying the packet containing the sequence number for the last byte of the response. When the response is chunked [10], sequence number matching cannot be used. Instead, ksniffer follows the chunk chain within the response body across multiple packets to determine the packet containing the last byte of the response. For CGI responses over HTTP 1.0 which do not specify the *Content-length:* field, the server closes the connection to indicate the end of the response. In this case, ksniffer simply keeps track of the time for the last data packet before the connection is closed.

ksniffer sets $w_1^{j,1}$'s end time to the arrival time of each response packet, plus $1/2 RTT$ to account for the transit time of the packet from server to client. ksniffer also sets pv_1^j 's end time to $w_1^{j,1}$'s end time. The end time will

monotonically increase until the server reply has been completed, at which point the (projected) end time will be equal to $t_k + .5RTT$, as shown in Figure 3. When ksniffer captures the last byte of the response at time t_k , $w_1^{j,1}$ is moved from F_1^j 's request queue to F_1^j 's finish queue, where it remains until either F_1^j is closed or until ksniffer determines that all segment retransmissions (if any) have been accounted for, which is discussed in Section 4.

Most Web browsers in use today serialize multiple HTTP requests over a connection such that the next HTTP request is not sent until the response for the previous request has been fully received. For these clients, there is no need for each flow object to maintain a queue of requests since there will only be one outstanding request at any given time. The purpose of ksniffer's request queue mechanism is to support HTTP pipelining, which has been adopted by a small, but potentially growing number of Web browsers. Under HTTP pipelining, a browser can send multiple HTTP requests at once, without waiting for the server to reply to each individual request. ksniffer's request queues provide support for HTTP pipelining by conforming to RFC2616 [10], which states that a server must send its responses to a set of pipelined requests in the same order that the requests are received. Since TCP is a reliable transport mechanism, requests that are pipelined from the client, in a certain order, are always received by the server in the same order. Any packet reordering that may occur in the network is handled by TCP at the server. ksniffer provides similar mechanisms to handle packet reordering so that HTTP requests are placed in F_1^j 's request queues in the correct sequence. This entails properly handling a packet that contains multiple HTTP requests as well as an HTTP request which spans packet boundaries.

At this point in time, ksniffer has properly determined $t_k + .5RTT$, the time at which the packet containing the last byte of data for $w_1^{j,1}$ was received by client C_j . If the Web page has no embedded objects then this marks the end of the pageview transaction. For example, if $w_1^{j,1}$ corresponds to a PDF file instead of an HTML file, ksniffer can determine that the transaction has completed, since a PDF file cannot have embedded objects.

If $w_1^{j,1}$ can potentially embed one or more Web objects, ksniffer cannot assume that pv_1^j has completed. Instead, it needs to determine what embedded objects will be downloaded to calculate the pageview response time. At time $t_k + .5RTT$, ksniffer cannot determine yet if requests for embedded objects are forthcoming or not. In particular, ksniffer does not parse the HTML within the container page to identify which embedded objects may be requested by the browser. Such processing is too computationally expensive for an online, high bandwidth system, and often does not even provide the necessary infor-

mation. For example, a JavaScript within the container page could download an arbitrary object that could only be detected by executing the JavaScript, not just parsing the HTML. Furthermore, HTML parsing would not indicate which embedded objects are directly downloaded from the server, since some may be obtained via caches or proxies. ksniffer instead takes a simpler approach based on waiting and observing what further HTTP requests are sent by the client, then using HTTP request header information to dynamically learn which container pages embed which objects.

3.4 Online Embedded Pattern Learning

ksniffer learns which container pages embed which objects by tracking the *Referer:* field in HTTP request headers. The *Referer:* field contained in subsequent requests is used to group embedded objects with their associated container page. Since the *Referer:* field is not always present, ksniffer develops *patterns* from those it does collect to infer embedded object relationships when requests are captured that do not contain a *Referer:* field. This technique is faster than parsing HTML, executing JavaScript, or walking the Web site with a Web crawler. In addition, it allows ksniffer to react to changes in container page composition as they are reflected in the actual client transactions.

ksniffer creates referer patterns on the fly. For each HTTP request that is captured, ksniffer parses the HTTP header and determines if the *Referer:* field is present. If so, this relationship is saved in a *pattern* for the container object. For example, when monitoring ibm.com, if a GET request for *obj1.gif* is captured, and the *Referer:* field is found to contain "www.ibm.com/index.html", ksniffer adds *obj1.gif* as an embedded object within the pattern for *index.html*. If a *Referer:* field is captured which specifies a host not being monitored by ksniffer, such as "www.xyz.com/buy.html", it is ignored.

ksniffer uses file extensions as a heuristic when building patterns. Web objects with an extension such as .ps and .pdf cannot contain embedded objects, nor can they be embedded within a page. As such, patterns are not created for them, nor are they associated with a container page. Web objects with an extension such as .gif or .jpg are usually associated with a container page, but cannot themselves embed other objects. Web objects with an extension such as .html or .htm can embed other objects or be embedded themselves. Each individual .html object has its own unique pattern, but currently an .html object is never a member of another object's pattern. This prevents cycles within the pattern structures, but results in ksniffer treating frames of .html pages as separate pageviews.

Taking this approach means that ksniffer does not need to be explicitly told which Web pages embed which ob-

jects – it learns this on its own. Patterns are persistently kept in memory using a hash table indexed by the container page URL. Each pv_i^j and container $w_k^{j,i}$ is linked to the pattern for the Web object it represents, allowing ksniffer to efficiently query the patterns associated with the set of active pageview transactions.

Since Web pages can change over time, patterns get dynamically updated, based on the client activity seen at the Web site. Therefore, a particular embedded object, *obj1.jpg*, may not belong to the pattern for container *index.html* at time t_i , and yet belong to the pattern at time t_{i+k} . Likewise, a pattern may not exist for *buy.html* at time t_i , but then be created at a later time t_{i+k} , when a request is captured. Of course, the same embedded object, *obj1.jpg*, may appear in multiple patterns, *index.html* and *buy.html*, at the same time or at different times. Since patterns are only created from client transactions, the set of patterns managed by ksniffer may be a subset of all the container pages on the Web site. This can save memory: ksniffer maintains patterns for container pages that are being downloaded, but not for those container pages on the Web site which do not get requested.

Only the *Referer*: field is used to manipulate patterns, and the embedded objects within a pattern are unordered. ksniffer places a configurable upper bound of 100 embedded objects within a pattern so as to limit storage requirements. When the limit is reached, an LRU algorithm is used for replacement, removing the embedded object which has not been linked to the container page in an HTTP request for the longest amount of time.

Each pattern typically contains a superset of those objects which the container page actually embeds. As the pattern changes, the new embedded objects get added to the pattern; but the old embedded objects only get removed from the pattern if the limit is reached. This is perfectly acceptable since ksniffer does not use patterns in a strict sense to determine, absolutely, whether or not a container page embeds a particular object.

Most Web browsers, including Internet Explorer and Mozilla, provide referer fields, but some do not and privacy proxies may remove them. To see what percentage of embedded objects have referer fields in practice, we analyzed the access log files of a popular musician resource Web site that has over 800,000 monthly visitors. The access logs covered a 15 month period from January 2003 until March 2004. 87% of HTTP requests had a referer field, indicating that a substantial portion of embedded objects may have referer fields in practice. ksniffer is specifically designed for monitoring high speed links that transmit a large number of transactions per second. In the domain of pattern generation, this is an advantage. The probability that at least one HTTP request with the *Referer*: field set for a particular container page will arrive within a given time interval is extremely high.

3.5 Embedded Object Processing

If a container page references embedded objects, the end of the transaction will be indicated by the packet containing the sequence number of the last byte of data, for the last object to complete transmission. To identify this packet, ksniffer determines which embedded object requests are related to each container page using the *Referer*: field of HTTP requests, file extension information, and the referer patterns discussed in Section 3.4.

In our example, suppose *index.html* contains references to five embedded images *obj1.gif*, *obj2.gif*, *obj3.gif*, *obj4.gif*, and *obj8.gif*. The embedded objects will be identified and processed as shown in Figure 4 (ignoring for the moment F_3^j). At time $t_k + .5RTT$, the browser parses the HTML document and identifies any embedded objects. If embedded objects are referenced within the HTML, the browser opens an additional connection, F_2^j , to the server so that multiple HTTP requests for the embedded objects can be serviced, in parallel, to reduce the overall latency of the transaction. The packet containing the sequence number of the last byte of the last embedded object to be fully transmitted indicates the end of the pageview transaction, t_e .

The start and end times for embedded object requests are determined in the same manner as previously described in Sections 3.2 and 3.3. Each embedded object that is requested is tracked in the same manner that the container page, *index.html*, was tracked. For example, when the second connection is initiated, ksniffer creates a flow object F_2^j to track the connection, and associates it with C_j . When the request for *obj1.gif* on F_2^j is captured at time t_q , a $w_1^{j,2}$ object is created for tracking the request, and is placed onto F_2^j 's request queue.

To determine the pageview response time, which is calculated as $t_e - t_0$, requires correlating embedded objects to their proper container page, which involves tackling a set of challenging problems. Clients, especially proxies, may be downloading multiple pageviews simultaneously. It is possible for a person to open two or more browsers and connect to the same Web site, or for a proxy to send multiple pageview requests to a server, on behalf of several remote clients. In either case, there can be multiple currently active pageview transactions simultaneously associated with the remote client C_j (e.g., pv_1^j , $pv_2^j \dots pv_k^j$). In addition, some embedded objects being requested may appear in multiple pageviews, and some Web objects may be retrieved from caches or CDNs. ksniffer applies a set of heuristics that attempt to determine the true container page for each embedded object. We present experimental results in Section 5 demonstrating that these heuristics are effective for accurately measuring client perceived response time.

For example, suppose that F_3^j in Figure 4 depicts

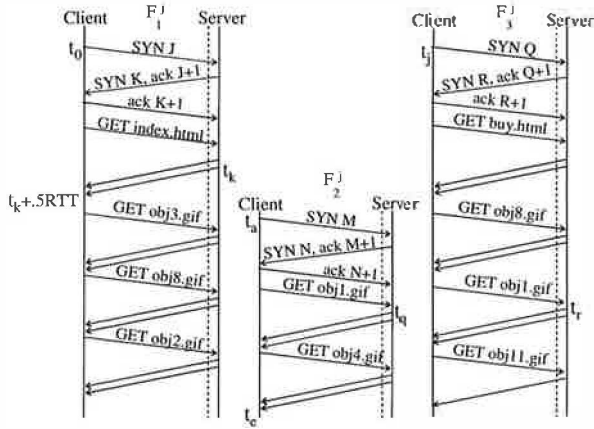


Figure 4: Downloading multiple container pages and embedded objects over multiple connections.

client C_j downloading *buy.html* at roughly the same time as *index.html* (i.e., $t_0 \approx t_j$). Suppose also that ksniffer knows in advance that *index.html* embeds $\{obj1.gif, obj3.gif, obj8.gif, obj4.gif, obj2.gif\}$ and that *buy.html* embeds $\{obj1.gif, obj8.gif, obj11.gif\}$. This means that both container pages are valid candidates for the true container page of *obj1.gif*. Whether or not $t_r < t_q$ is a crucial indication as to the true container page. At time t_a , when connection F_2^j is being established, there is no information which could distinguish whether this connection belongs to *index.html* or *buy.html*. The only difference between F_1^j , F_2^j and F_3^j with respect to the TCP/IP 4-tuple is the remote client port number. Hence only the client, C_j , can be identified at time t_a , and at time t_q , it is unknown whether *index.html* or *buy.html* is the true container page for *obj1.gif*.

To manage pageviews and their associated embedded objects, ksniffer maintains three lists of active pageviews for each client, each sorted by request time, as shown in Figure 5. The *loners queue* contains pageviews which represent objects that cannot have embedded objects. These pageviews are kept in their own list, which is never searched when attempting to locate a container page for a new embedded object request. All other pageviews, which could potentially embed an object, are placed on both a *FIFO pageview queue* and the *pageview hash table*. This enables ksniffer to quickly locate the youngest candidate container page. Each pageview also maintains an embedded object hash table, not shown in Figure 5, that consists of the embedded objects associated with that pageview and state indicating whether and to what extent they have been downloaded.

Given a request $w_i^{j,k}$ captured on flow F_k^j for client C_j , ksniffer will perform the following actions:

1. If $w_i^{j,k} \in \{.html, .shtml, \dots\}$ ksniffer will treat $w_i^{j,k}$ as a container page by placing it into the pageview hash table (and FIFO queue) for client C_j . In addition, if a pageview is currently associated with F_k^j ,

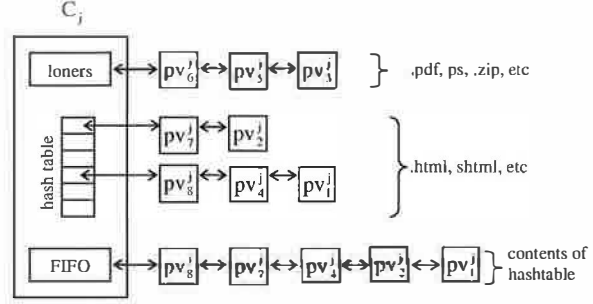


Figure 5: Client active pageviews.

ksniffer assumes that pageview is done.

2. If $w_i^{j,k} \in \{.pdf, .ps, \dots\}$ ksniffer will treat $w_i^{j,k}$ as a loner object by placing it on the loner queue for C_j . In addition, if a pageview is currently associated with F_k^j , ksniffer assumes it is done.
3. If $w_i^{j,k} \in \{.jpg, .gif, \dots\}$ then
 - (a) If the *Referer:* field contains the monitored server name, such as *www.ibm.com/buy.html*, then C_j 's pageview hash table is searched to locate pv_c^j , the youngest pageview downloading that container page (*buy.html*) that has yet to download $w_i^{j,k}$. If pv_c^j exists, $w_i^{j,k}$ is associated to pv_c^j as one of its embedded objects. If no pageview meets the criterion, pv_c^j is created and $w_i^{j,k}$ is associated to it.
 - (b) If the *Referer:* field contains a foreign host name, such as *www.xyz.com/buy.html*, then $w_i^{j,k}$ is treated as a loner object.
 - (c) If $w_i^{j,k}$ has no *Referer:* field, then the FIFO queue is searched to locate, pv_c^j , the youngest pageview which has $w_i^{j,k}$ in its referer pattern and has yet to download $w_i^{j,k}$. If pv_c^j exists, then $w_i^{j,k}$ is associated to pv_c^j as one of its embedded objects. If no pageview meets the criterion, then $w_i^{j,k}$ is treated as a loner object.

The algorithm above is based on several premises. If a request for an embedded object $w_i^{j,k}$ arrives with a referer field containing the monitored server as the host (e.g., *www.ibm.com/buy.html*), then the remote browser almost certainly must have previously downloaded that container page (e.g., *buy.html*) from the monitored server (e.g., *www.ibm.com*), parsed the page, and is now sending the request for the embedded object $w_i^{j,k}$. If ksniffer failed to capture the request for the container page (e.g., *buy.html*) it is highly likely that it is being served from the browser cache for this particular transaction. If a request for an embedded object arrives with a referer field containing a foreign host (e.g., *www.xyz.com/buy.html*), it is highly likely that the foreign host is simply embedding objects from the monitored Web site into its own pages.

When a request for an embedded object arrives with-

out a referer field, every pageview associated with the client becomes a potential candidate for the container page of that object. This is depicted in Figure 4 when the request for *obj1.gif* arrives without a *Referer:* field. If the client is actually a remote proxy, then the number of potential candidates may be large. ksniffer applies the patterns described in Section 3.4 as a means of reducing the number of potential candidates and focusing on the true container page of the embedded object. The heuristic is to locate the youngest pageview which contains the object in its pattern, but has yet to download the object. Patterns are therefore exclusionary. Any candidate pageview not containing the embedded object in its pattern is excluded from consideration. This may result in the true container page being passed over, but as mentioned in Section 3.4, the likelihood that a container page embeds an object that does not appear in the page's pattern is very low for an active Web site. If a suitable container pageview is not found, then the object is treated as a loner object. If a *Referer:* field is missing, then most likely it was removed by a proxy and not a browser on the client machine; but if the proxy had cached the container page during a prior transaction, it is likely to have cached the embedded object as well. This implies the object is not being requested as part of a page, but being downloaded as an individual loner object.

If a client downloads an embedded object, such as *obj1.gif*, it is unlikely that the client will download the same object again, for the same container page. If an object appears multiple places within a container page, most browsers will only request it once from the server. Therefore, ksniffer not only checks if an embedded object is in the pattern for a container page, but also checks if that instance has already downloaded the object or not.

The youngest candidate is usually a better choice than the oldest candidate. If browsers could not obtain objects from a cache or CDN, then the oldest candidate would be a better choice, based on FCFS. Since this is not the case choosing the oldest candidate will tend to assign an object *obj1.jpg* to a container page whose 'slot' for *obj1.jpg* was already filled via an unseen cache hit. This tends to overestimate response time for older pages. It is more likely that an older page obtained *obj1.jpg* from a cache and that the younger page is the true container for *obj1.jpg*, than vice versa.

ksniffer relies on capturing the last byte of data for the last embedded object to determine the pageview response time. However, given the use of browser caches and CDNs, not all embedded objects will be seen by ksniffer since not all objects will be downloaded directly from the Web server. The purpose of a cache or CDN is to provide much faster response time than can be delivered by the original Web server. As a result, it is likely that objects requested from a cache or CDN will be received by the

client before objects requested from the original server. If the Web server is still serving the last embedded object received by the client, other objects served from a cache or CDN will not impact ksniffer's pageview response time measurement accuracy. If the last embedded object received by the client is from a cache or CDN, ksniffer will end up not including that object's download time as part of its pageview response time. Since caches and CDNs are designed to be fast, the time unaccounted for by ksniffer will tend to be small even in this case.

Given that embedded objects may be obtained from someplace other than the server, and that a pattern for a container page may not be complete, how can ksniffer determine that the last embedded object has been requested? For example, at time t_e , how can ksniffer determine whether the entire download for *index.html* is completed, or another embedded object will be downloaded for *index.html* on either F_1^j or F_2^j ? This is essentially the same problem described at the end of Section 3.3 with respect to whether or not a embedded objects requests will follow a request for a container page or not.

ksniffer approaches this problem in two ways. First, if no embedded objects are associated to a pageview after a timeout interval, the pageview transaction is assumed to be complete. A six second timeout is used by default, in part based on the fact that the current ad hoc industry quality goal for complete Web page download times is six seconds [19]. If a client does not generate additional requests for embedded objects within this time frame, it is very likely that the pageview is complete. ksniffer also cannot report the response time for a pageview until the timeout expires. A six second timeout is small enough to impose only a modest delay in reporting.

Second, if a request for a container page, $w_k^{j,i}$, arrives on a persistent connection F_i^j , then we consider that all pageview transactions associated with each prior object, $w_b^{j,i}$, $b < k$, on F_i^j to be complete. In other words, a new container page request over a persistent connection signals the completion of the prior transaction and the beginning of a new one. We believe this to be a reasonable assumption, including under pipelined requests, since in most cases, only the embedded object requests will be pipelined. Typical user behavior will end up serializing container page requests over any given connection. Hence, the arrival of a new container page request would indicate a user click in the browser associated with this connection. Taking this approach also allows ksniffer to properly handle quick clicks, when the user clicks on a visible link before the entire pageview is downloaded and displayed in the browser.

4 Packet Loss

Studies have shown that the packet loss rate within the Internet is roughly 1-3% [34]. We classify packet loss

into three types: A) a packet is dropped by the network before being captured by ksniffer, B) a packet is dropped by the network after being captured and C) a packet is dropped by the server or client after being captured. Types A and B are most often due to network congestion or transmission errors while type C drops occur when the Web server (or, less likely, the client) becomes temporarily overloaded. The impact that a packet drop has on measuring response time depends not only on where or why it was dropped, but also on the contents of the packet. We first address the impact of SYN drops, then look at how a lost data packet can affect response time measurements.

Figure 3 depicts the well known TCP connection establishment protocol. Suppose that the initial SYN which is transmitted at time t_0 is either dropped in the network or at the server. In either case, no SYN/ACK response is forthcoming from the server. The client side TCP recognizes such SYN drops through use of a timer [27]. If a response is not received in 3 seconds, TCP will retransmit the SYN packet. If that SYN packet is also dropped by the network or server, TCP will again resend the same SYN packet, but not until after waiting an additional 6 seconds. As each SYN is dropped, TCP doubles the wait period between SYN retransmissions: 3 s, 6 s, 12 s, 24 s, etc. TCP continues in this manner until either the configured limit of retries is reached, at which time TCP reports “unable to connect” back to the browser, or the user takes an action to abort the connection attempt, such as refreshing or closing the browser.

This additional delay has a large impact on the client response time. Suppose there is a 3% network packet loss rate from client to server. Three percent of the SYN packets sent from the remote clients will be dropped in the network before reaching ksniffer or the server. The problem is that since the SYN packets are dropped in the network before reaching the server farm, both ksniffer and the server are completely unaware that the SYNs were dropped. This will automatically result in an error for any traffic monitoring system which measures response time using only those packets which are actually captured. If each client is using two persistent connections to access the Web site, this error will be 180% for a 100 ms response time and a 4.5% error for a 4s response time. Under HTTP 1.0 without Keep-Alive, where a connection is opened to obtain each object, the probability of a network SYN drop grows with the number of objects in the pageview. For a page download of 10 objects, there is a 30% chance of incurring the 3 second retransmission delay, a 60% chance for 20 objects and a 90% chance for 30 objects.

ksniffer uses a simple technique for capturing this undetectable connection delay (type ‘A’ SYN packet loss). Three counters are kept for each subnet. One of the three

counters is incremented whenever a SYN/ACK packet is retransmitted from the server to the client (which indicates that the SYN/ACK packet was lost in the network). The counter that gets incremented depends on how many times the SYN/ACK has been transmitted. Every time a SYN/ACK is sent twice, the first counter is incremented, every time a SYN/ACK packet is sent 3 times, the second counter is incremented, and every time a SYN/ACK is sent 4 times, the third counter is incremented. Whenever a SYN packet arrives for a new connection, if one of the three counters is greater than zero, then ksniffer subtracts the appropriate amount of time from the start time of the connection and decrements the counter (round robin is used to break ties). Assuming that a SYN packet will be dropped as often as a SYN/ACK, this gives ksniffer a reasonable estimate for the number of connections which are experiencing a 3 s, 9 s, or 21 s connection delay.

The same retransmission delays are incurred when SYNs are dropped by the server (type ‘C’). In this case, ksniffer is able to capture and detect that the SYNs were dropped by the server, and distinguish these connection delays, which are due to server overload, from those previously described, which are due to network congestion. ksniffer also determines when a client is unable to connect to the server. If the client reattempts access to the Web site in the next six seconds after a connection failure, ksniffer considers the time associated with the first failed connection attempt as part of the connection latency for the reattempt; otherwise the failed connection attempt is reported under the category “frustrated client”.

Similar undetected latency occurs when a GET request is dropped in the network before reaching ksniffer or the server, then retransmitted by the client. An undetected GET request drop differs from an undetected SYN drop in two ways. First, unlike SYN drops, TCP determines the retransmission timeout period based on RTT and a number of implementation dependent parameters. ksniffer implements the standard RTO calculation [28] using Linux TCP parameters, and adjusts for this undetectable time in the same manner as mentioned above. Second, a dropped GET request will only affect the measurement of the overall pageview response time if the GET request is for a container page and is not the first request over the connection. Otherwise, the start of the transaction will be indicated by the start of connection establishment, not the time of the container page request.

As mentioned earlier, ksniffer often expects to capture the packet containing the sequence number of the last byte of data for a particular request. To capture retransmissions, ksniffer uses a timer along with the finish queue on each flow to capture retransmitted packets and update the end of response time appropriately. Suppose the last packet of a response is captured by ksniffer at time t_k , at which point ksniffer identifies it as containing

the sequence number for the last byte of the response, and moves the $w_k^{j,i}$ request object from the flow's request queue to the flow's finish queue. The packet is then dropped in the network before reaching the client (type 'B'). At time t_{k+h} , ksniffer will capture the retransmitted packet and, using its sequence number, determine that it is a retransmission for $w_k^{j,i}$, which is located on the finish queue. The completion time of $w_k^{j,i}$ is then set to the timestamp of this packet.

5 Experimental Results

We implemented ksniffer as a set of Linux kernel modules and installed it on a commodity PC to demonstrate its accuracy and performance under a wide range of Web workloads. We report an evaluation of ksniffer in a controlled experimental setting as well as an evaluation of ksniffer tracking user behavior at a live Internet Web site.

Our experimental testbed is shown in Figure 6. We used a traffic model based on Surge [3] but made some minor adjustments to reflect more recent work [14, 31] done on characterizing Web traffic: the maximum number of embedded objects in a given page was reduced from 150 to 100 and the percentage of base, embedded, and loner objects were changed from 30%, 38% and 32% to 42%, 48% and 10%, respectively. The total number of container pages was 1041, with 959 unique embedded objects. 49% of the embedded objects are embedded by more than one container page. We also fixed a bug in the modeling code and included CGI scripts in our experiments, something not present in Surge.

For traffic generation, we used an updated version of WaspClient [25], which is a modified version of the client provided by Surge. Virtual clients on each machine cycle through a series of pageview requests, first obtaining the container page then all its embedded objects. A virtual client can open 2 parallel TCP connections for fetching pages, mimicking the behavior of Microsoft IE. Requests on a TCP connection are serialized, so that the next request is not sent until the current response on that connection is obtained. In addition, each virtual client binds to a unique IP address using IP aliasing on the client machine. This lets each client machine appear to the server as a collection of up to 200 unique clients from the same subnet.

To emulate wide-area conditions, we extended the rshaper [30] bandwidth shaping tool to include packet loss and round trip latencies. We installed this software on each client traffic generator machine, enabling us to impose packet drops as well as the RTT delays between 20 to 200 ms as specified in Figure 6.

To quantify the accuracy of the client perceived response times measured by ksniffer, we ran fifteen different experiments with different traffic loads under non-ideal and high-stress operating conditions and compared

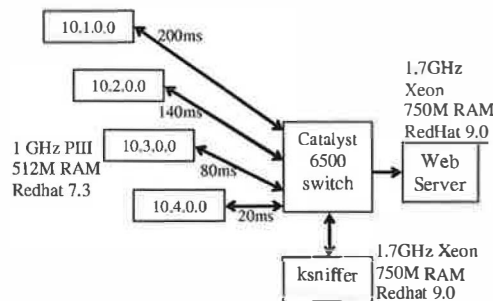


Figure 6: Experimental environment.

ksniffer's measurements against those obtained by the traffic generators executing on the client machines. We measured with two different Web servers, Apache and TUX, used both HTTP 1.0 without Keep-Alive and persistent HTTP 1.1, and included a combination of static pages and CGI programs for Web content. We also measured in the presence of network and server packet loss, missing referer fields, client caching, and near gigabit traffic rates. Table 1 summarizes these experimental results. In all cases, the difference between the mean response time as determined by ksniffer, and that measured directly on the remote client was less than 5%. Furthermore, the absolute time difference between ksniffer and client-side instrumentation was in some cases less than 1 ms and in all cases less than 50 ms.

All tests (except Tests S1 and S2) were done under non-ideal conditions found in the Internet with 2% packet loss and 20% missing referer fields. Each client requested the same sequence of pageviews, but since each traffic generator machine was configured with a different RTT to the Web server as shown in Figure 6, the clients took different amounts of time to obtain all of their pages, resulting in a variable load on the Web server over time. For example, Figure 7 shows results from Test F comparing ksniffer against client-side instrumentation in measuring pageviews/s over time. There are two lines in the figure, but they are hard to distinguish because ksniffer's pageview count is so close to direct client-side instrumentation. Figure 8 shows results from Test F comparing ksniffer against client-side instrumentation in measuring mean client perceived pageview response time for each 1 second interval. ksniffer results are very accurate and hard to distinguish from client-side instrumentation. As indicated by Figure 7, the variable response time is due to the completion of clients. During the initial 250 s, clients from each of the four subnets are actively making requests. At around 250 s, the clients from subnet 10.4.0.0 with RTT 20 ms have completed, while clients from the other subnets remain active. At around 300 s, the clients from subnet 10.3.0.0 with RTT of 80 ms have completed, leaving clients from subnets 10.2.0.0 and 10.1.0.0 active. At time 475 s, clients from subnet 10.2.0.0 with RTT of 140 ms have completed, leaving only those clients from subnet 10.1.0.0 with RTT

	Virtual Clients	Web Server	HTTP	PV/s	URL/s	Mbps	Client RT	ksniffer RT	diff (ms)	% diff	elapsed time
A	120	Apache	1.0	5-140	5-625	1-60	1.528s	1.498s	-29	-1.9	133m
B	120	Apache	1.0	5-160	10-660	1-60	1.513s	1.483s	-30	-2.0	133m
C	120	Apache	1.1	10-180	30-730	3-70	1.003s	0.981s	-22	-2.2	79m
D	120	Apache	1.1	10-400	40-1520	3-140	0.726s	0.699s	-27	-3.7	72m
E	800	TUX	1.0	65-750	260-3000	15-270	1.556s	1.506s	-49	-3.2	20m
F	800	TUX	1.1	125-1370	500-5300	35-455	0.815s	0.782s	-33	-4.1	11m
G	500	Apache	1.0	35-500	140-2000	10-200	1.537s	1.489s	-48	-3.1	32m
H	400	Apache	1.1	60-690	250-2880	15-250	0.792s	0.825s	-33	-4.0	22m
I	500	Apache	1.1	60-700	260-3000	20-265	0.884s	0.929s	-45	-4.8	18m
S1	16	TUX	1.0	1909	8,007	690	7.8ms	7.7ms	-0.17	-2.2	210s
S2	80	TUX	1.1	2423	10,164	878	30.5ms	29.7ms	-0.83	-2.7	165s
V	800	TUX	1.0	0-2410	0-10,000	0-850	0.574s	0.571s	-3	-0.5	29m
O1	800	Apache	1.0	419	1756	152	1.849s	1.806s	-42	-2.3	16m
O2	240	Apache	1.1	728	3054	264	.328s	.318s	-10	-3.1	9m
X	800	Apache	1.0	2174	9120	462	.365s	.363s	-1.7	-0.5	184s

Table 1: Summary of results.

subnets	RTT (ms)	Client RT	ksniffer RT	diff (ms)	% diff	ksniffer RTT(ms)
10.1.0.0	200	1.424s	1.391s	-33	-2.3	199.8
10.2.0.0	140	1.099s	1.073s	-26	-2.4	139.8
10.3.0.0	80	0.824s	0.806s	-18	-2.3	79.7
10.4.0.0	20	0.666s	0.656s	-10	-1.6	19.9

Table 2: Mean RT per subnet, Test C.

of 200 ms. Note that, although the pageview request rate decreases, the mean response time increases because the remaining clients have larger RTTs to the Web server and thus incur larger response times.

Table 2 shows results for Test C obtained by implementing a longest prefix matching algorithm based on [5] in ksniffer to categorize RTT and response time on a per subnet basis. These results show that ksniffer provides accurate pageview response times as compared to client-side instrumentation even on a per subnet basis when different subnets have different RTTs to the Web server. ksniffer RTT measurements are also very accurate as compared to the actual RTT used for each subnet. The results show how this mechanism can be very effective in differentiating performance and identifying problems across different subnets.

Tests S1 and S2 were done under high bandwidth conditions to show results at the maximum bandwidth rate possible in our testbed. This was done by using the faster TUX Web server and by imposing no packet loss or network delay. For HTTP 1.1, 80 virtual clients generated the greatest bandwidth rate, but under HTTP 1.0 only 16 clients generated the highest bandwidth rate. ksniffer is within 3% of client-side measurements, even under rates of 690 Mbps and 878 Mbps of HTTP content. The absolute time difference between ksniffer and client response time measurements was less than 1 ms. We note that the

resolution of the packet timer on ksniffer is only 1 ms, due to the Linux clock timer granularity. Under HTTP 1.0 without Keep-Alive, each object retrieved requires its own TCP connection. The TCP connection rate under Test S1 was 8,000 connections/s. The results demonstrate ksniffer's ability to track TCP connection establishment and termination at high connection rates.

Test V was done with severe variations in load alternating between no load and maximum bandwidth load by switching the clients between on and off modes every 50 s. Figure 9 compares ksniffer response time with that measured at the client, and Figure 10 compares the distribution of the response time. This indicates ksniffer's accuracy under extreme variations in load.

Tests O1 and O2 were done with the Web server experiencing overload and therefore dropping connections. We configured Apache to support up to 255 simultaneous connections, then started 240 virtual clients. Since each client opens two connections to the server to obtain a container page and its embedded objects, this overwhelmed Apache. During Test O1 and O2, the Web server machine reported a connection failure rate of 27% and 12%, respectively. Table 1 shows that ksniffer's pageview response time for these tests were only 3% less than those from the client-side. These results show ksniffer's ability to measure response times accurately in the presence of both server overload and network packet loss.

Test X was done to show ksniffer performance with caching clients by modifying the clients so that 50% of the embedded objects requested were obtained from a zero latency local cache. Figure 11 compares ksniffer and client-side instrumentation in measuring pageview response time over the course of the experiment. The results show that ksniffer can provide very accurate response time measurements in the presence of client

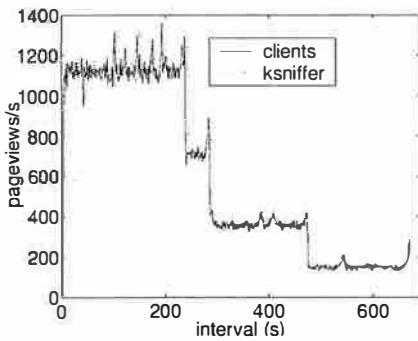


Figure 7: Test F, pageviews.

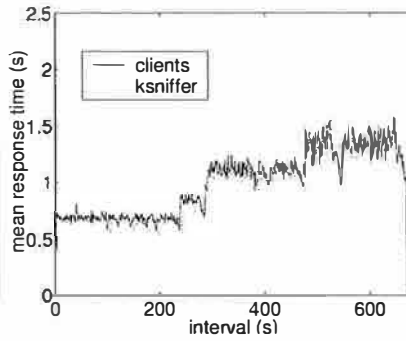


Figure 8: Test F, response time.

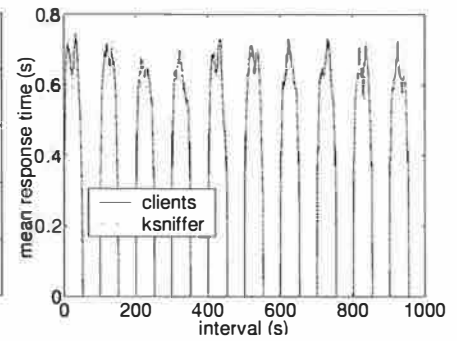


Figure 9: Test V, response time.

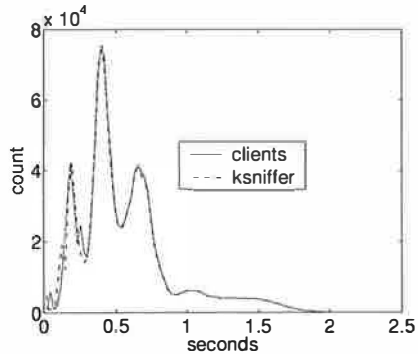


Figure 10: Test V, RT distribution.

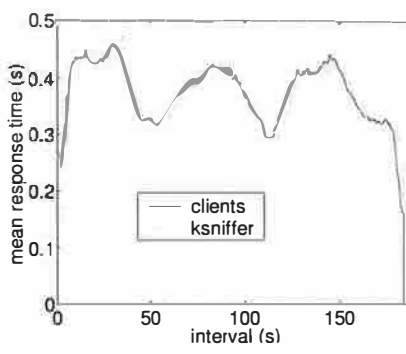


Figure 11: Text X, response time.

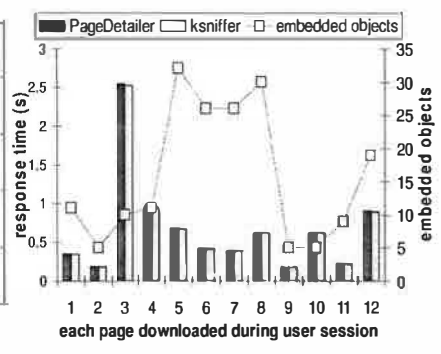


Figure 12: Live Internet Web site.

caching as well.

We deployed ksniffer in front of a live Internet Web site, GuitarNotes.com, which is hosted in NYC. Figure 12 depicts results for tracking a single user during a logon session from Hawthorne, NY. Using MS IE V6, and beginning with an empty browser cache, the user first accessed the home page and then visited a dozen pages within the site including the product review section, discussion forum, FAQ, classified ads, and performed several site searches for information. This covered a range of static and dynamically generated pageviews. The number of embedded objects for each page varied between 5 and 30, and is indicated by the dotted line, which is graphed against the secondary Y axis on the right. These objects included .gif, .css and .js objects.

PageDetailer [15] was executing on the client machine monitoring all socket level activity of IE. PageDetailer uses a Windows socket probe to monitor and timestamp each socket call made by the browser: connect(), select(), read() and write(). By parsing the HTTP requests and replies, it is able to determine the response time for a pageview, as well as for each embedded object within a page. The pageview response time is calculated as the difference between the connect() system call entry and the return from the read() system call for the last byte of data of the last embedded object. As shown in Figure 12, the response time which ksniffer calculates in NYC at the Web server is nearly identical to that measured by PageDetailer running on the remote client machine. For each of the twelve pages downloaded by the client,

ksniffer is within 5% of the response time recorded by PageDetailer.

ksniffer provides excellent performance scalability compared to common user-space passive packet capture systems. Almost all existing passive packet capture systems in use today are based on libpcap [33]. Libpcap is a user space library that opens a raw socket to provide packets to user space monitor programs. As a scalability test, we wrote a libpcap based traffic monitor program whose only function was to count TCP packets. Executing on the same physical machine as ksniffer, the libpcap packet counter program began to drop a large percentage of packets when the traffic rate was roughly 325 Mbps. In contrast, ksniffer performs complex pageview analysis at near gigabit traffic rates without such packet loss.

6 Related Work

There are a number of approaches currently being taken to address the problem of obtaining response time in the context of Web services. A number of companies [8, 20, 24, 32] provide active probing of a Web site by periodically measuring response times at a geographically distributed set of monitors. There are several limitations with this approach. First, no real Web traffic by the actual clients is measured; only the response time for transactions generated by the monitors are reported. Second, any approach based on coarsened-grained sampling may suffer from statistical biases. Third, monitors are limited to performing transactions that do not affect other users or modify state in backend databases. For exam-

ple, it would be unwise to configure a monitor to actually purchase an airline ticket or trade stock on an open exchange. Fourth, the information gathered by monitors is generally not available at the Web server in real-time, limiting the ability of a Web server to respond to changes in response time to meet delay bound guarantees. Lastly, CDN providers are known to place servers near monitors used by these companies to artificially improve their own performance measurements [7].

A second approach involves instrumenting Web pages with client-side scripting that gathers client response time statistics [29]. This approach can be used to track actual client transactions. However, client-side scripting is a ‘post-connection’ approach and therefore does not account for delays due to TCP connection setup or waiting in kernel queues on the Web server, which can be significant when network and server resources are overloaded. Client-side scripting cannot be applied to non-HTML files that cannot be instrumented, such as PDF and Postscript files. It may also not work for older browsers or browsers with scripting capabilities disabled, such as mobile devices. Client browser measurements cannot accurately decompose the response time into server and network components, providing no insight into whether server or network providers are responsible for problems.

A third approach requires the Web server to track when requests arrive and complete service, either at the application-level [2, 18, 21, 22] or at the kernel-level [27]. This approach has the desirable properties that it only requires information available at the Web server and can be used for non-HTML content. However, application-level approaches do not account for network interactions or delays due to TCP connection setup or waiting in kernel queues on the Web server. Previous results demonstrate that application-level Web server measurements can under estimate response time by more than an order of magnitude [27]. Two of the authors of this paper previously developed Certes [27], a kernel-level approach that accounts for TCP connection setup time and time spent waiting in kernel queues in measuring response time at a per connection level. *ksniffer* extends this work by measuring response time per pageview without any modifications to the Web server.

A fourth approach is to simply log network packets to disk, and then use the log files to reconstruct the client response time [1, 4, 9, 11, 12]. This kind of analysis is performed offline, using multiple passes and limited to analyzing only reasonably sized log files [31]. *ksniffer*’s correlation algorithm differs from EtE [11] in that it does not require multiple passes and offline operation, uses file extensions and refer host names in addition to the filename in the refer field, handles multiple requests for the same Web page from the same client, and accounts

for connection setup time and packet loss in determining response time. [9] describes many of the issues involved in TCP/HTTP reconstruction, but does not consider the problem of measuring response time.

Other approaches exist which can provide mechanisms for filtering and analyzing packet traces online, such as GigaScope [6], Nprobe [12], NetQoS [26], libpcap [33], and BPF [23]. However, these systems do not provide any higher-level functionality to determine pageview response times from live Web traffic. Most of this work has focused on improving packet filtering performance, which is not particularly applicable when all traffic into and out of a Web server is of interest, rather than a narrow subset.

Note that *ksniffer* shares certain limitations that are present in all network traffic monitors. Response time components due to processing on the remote client machines cannot be directly measured from server-side network traffic. Examples include times for DNS query resolution and HTML parsing and rendering on the client. Embedded objects obtained from locations other than the monitored servers may have an impact on accuracy as well, but only if their download completion time exceeds that of the last object obtained from the monitored server.

7 Conclusions and Future Work

We have designed, implemented and evaluated *ksniffer*, a kernel-based traffic monitor that can be colocated with Web servers to measure their performance as perceived by remote clients in real-time. As a passive network monitor, *ksniffer* requires no changes to clients or Web servers, and does not perturb performance in the way that intrusive instrumentation methods can. *ksniffer* determines client perceived pageview response times using novel, online mechanisms that take a “look once, then drop” approach to packet analysis to reconstruct TCP connections and learn client pageview activity.

We have implemented *ksniffer* as a set of loadable Linux kernel modules and validated its performance using both a controlled experimental testbed and a live Internet Web site. Our results show that *ksniffer*’s in-kernel design scales much better than common user-space approaches, enabling *ksniffer* to monitor gigabit traffic rates using only commodity hardware, software, and network interface cards. More importantly, our results demonstrate *ksniffer*’s unique ability to accurately measure client perceived response times even in the presence of network and server packet loss, missing HTTP referer fields, client caching, and widely varying static and dynamic Web content.

Future work includes integrating *ksniffer* with a cluster management system and developing mechanisms that manage resources to achieve specified response time goals. Such a management system would base resource

allocation decisions on the response time as perceived by the remote client instead of the response time as reported by other means. This may raise some interesting scheduling and allocation problems, particularly in the context of resource constrained Web sites. We expect to combine machine learning techniques with models of TCP/IP and client behavior to achieve our goals.

8 Acknowledgements

This work was supported in part by NSF grants ANI-0117738 and ANI-0240525 and an IBM SUR Award. Our thanks to Srinu Seshan for his help with the rshaper traffic tool, and to Gong Su for sharing his system expertise.

References

- [1] M. K. Aguilera et al. Performance Debugging for Distributed Systems of Black Boxes. In *SOSP '03*, p. 74–89, Lake George, NY, October 2003.
- [2] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing Differentiated Levels of Service in Web Content Hosting. In *Workshop on Internet Server Performance (WISP)*, Madison, WI, 1997.
- [3] P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *ACM SIGMETRICS*, p. 151–160, Madison, WI, November 1998.
- [4] R. Caceres et al. Measurement and Analysis of IP Network Usage and Behavior. *IEEE Communications Magazine*, 38(5), May 2000.
- [5] T. Chiueh and P. Pradhan. High Performance IP Routing Table Lookup using CPU Caching. In *IEEE INFOCOMM*, v 3, p. 1421–1428, 1999.
- [6] C. Cranor et al. Gigascope: A Fast and Flexible Network Monitor. Technical Report TD-5ABQY6, AT&T Labs–Research, Floram Park, NJ, May 2002.
- [7] P. Danzig. Ideas for Next Generation Content Delivery. In *NOSSDAV 2001*, http://www.nossdav.org/2001/keynote_nossdav2001.ppt, Port Jefferson, NY, June 2001. ACM.
- [8] Exodus. <http://www.exodus.com/>.
- [9] A. Feldmann. BLT: Bi-layer Tracing of HTTP and TCP/IP. In *WWW-9*, p. 321–335, May 2000.
- [10] R. Fielding et al. Hypertext Transfer Protocol HTTP 1.1. In *IETF RFC 2616*. June 1999.
- [11] Y. Fu, L. Cherkasova, W. Tang, and A. Vahdat. EtE: Passive End-to-End Internet Service Performance Monitoring. In *USENIX 2002*, p. 115–130, Monterey, CA, June 2002.
- [12] J. Hall, I. Pratt, and I. Leslie. Non-intrusive Estimation of Web Server Delays. In *Proceedings of the IEEE Conference on Local Computer Networks (LCN)*, Tampa, Florida, November 2001.
- [13] J. Hay, W. Feng, , and M. Gardner. Capturing Network Traffic with a MAGNeT. In *5th Annual Linux Showcase and Conference (ALS)*, p. 61–70, Oakland, California, November 2001.
- [14] F. Hernandez-Campos, K. Jeffay, and F. D. Smith. Tracking the Evolution of Web Traffic: 1995–2003. In *MASCOTS 2003*, p. 16–25, Orlando, FL, October 2003.
- [15] IBM AlphaWorks. *Page Detailer*, <http://www.alphaworks.ibm.com/tech/pagedetailer>.
- [16] P. Joubert et al. High-Performance Memory-Based Web Servers: Kernel and User-Space Performance. In *USENIX 2001*, p. 175–188, Boston, MA, June 2001.
- [17] M. F. Kaashoek et al. Application Performance and Flexibility on Exokernel Systems. In *SOSP*, Saint-Malo, France, Oct. 1997.
- [18] V. Kanodia and E. Knightly. Multi-Class Latency-Bounded Web Services. In *IEEE/IFIP IWQoS*, Pittsburgh, PA, June 2000.
- [19] T. Keeley. Thin, High Performance Computing over the Internet. In *MASCOTS 2000*, p. 407, San Francisco, CA, Aug. 2000.
- [20] KeyNote. <http://www.keynote.com>.
- [21] K. Li and S. Jamin. A Measurement-Based Admission-Controlled Web Server. In *IEEE INFOCOMM*, p. 651–659, New York, NY, June 2002.
- [22] C. Lu, T. Abdelzaher, J. Stankovic, and S. H. Son. A Feedback Control Approach for Guaranteeing Relative Delays in Web Server. In *7th IEEE Real-Time Technology and Applications Symposium*, p. 51–62, Taipei, Taiwan, June 2001.
- [23] S. McCanne and V. Jacobson. The BSD Packet Filter: A new Architecture for User-Level Packet Capture. In *USENIX 1993*, p. 259–270, 1993.
- [24] Mercury Interactive. <http://www-heva.mercuryinteractive.com>.
- [25] E. Nahum, M. Rosu, S. Seshan, and J. Almeida. The Effects of Wide-Area Conditions on WWW Server Performance. In *ACM SIGMETRICS*, p. 257–267, Cambridge, MA, June 2001.
- [26] NetQoS. <http://www.NetQoS.com>.
- [27] D. Olshefski, J. Nieh, and D. Agrawal. Using Certes to Infer Client Response Time at the Web Server. *ACM Transactions on Computing Systems*, February 2004.
- [28] V. Paxson and M. Allman. Computing TCP's Retransmission Time. In *IETF RFC 2988*. November 2000.
- [29] R. Rajamony and M. Elnozahy. Measuring Client-Perceived Response Times on the WWW. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, San Francisco, CA, March 2001.
- [30] A. Rubini. rshaper. <http://www.linux.it/~rubini/software/index.html>.
- [31] F. D. Smith, F. H. Campos, K. Jeffay, and D. Ott. What TCP/IP Protocol Headers can tell us about the Web. In *ACM SIGMETRICS*, p. 245–256, Cambridge, MA, June 2001.
- [32] StreamCheck. <http://www.streamcheck.com>.
- [33] The libpcap project. <http://sourceforge.net/projects/libpcap>.
- [34] Y. Zhang, N. Duffield, V. Paxson, and S. Shenker. On the Constancy of Internet Path Properties. In *Proceedings of ACM SIGCOMM Internet Measurement Workshop (IMW)*, San Francisco, CA, November 2001.

FFPF: Fairly Fast Packet Filters

Herbert Bos[†], Willem de Bruijn[†], Mihai Cristea*, Trung Nguyen*, Georgios Portokalidis*

[†]*Vrije Universiteit Amsterdam, The Netherlands*

{herbertb, wdb}@few.vu.nl

^{*}*Universiteit Leiden, The Netherlands*

{cristea, tnguyen, gportoka}@liacs.nl

Abstract

FFPF is a network monitoring framework designed for three things: speed (handling high link rates), scalability (ability to handle multiple applications) and flexibility. Multiple applications that need to access overlapping sets of packets may share their packet buffers, thus avoiding a packet copy to each individual application that needs it. In addition, context switching and copies across the kernel boundary are minimised by handling most processing in the kernel or on the network card and by memory mapping all buffers to userspace, respectively. For these reasons, FFPF has superior performance compared to existing approaches such as BSD packet filters, and especially shines when multiple monitoring applications execute simultaneously. Flexibility is achieved by allowing expressions written in different languages to be connected to form complex processing graphs (not unlike UNIX processes can be connected to create complex behaviour using pipes). Moreover, FFPF explicitly supports extensibility by allowing new functionality to be loaded at runtime. By also implementing the popular `pcap` packet capture library on FFPF, we have ensured backward compatibility with many existing tools, while at the same time giving the applications a significant performance boost.

1 Introduction

Most network monitoring tools in use today were designed for low-speed networks under the assumption that computing speed compares favourably to network speed. In such environments, the costs of copying packets to user space prior to processing them are acceptable. In today's networks, this assumption is no longer true. The number of cycles available to process a packet before the next one arrives (the cycle budget) is minimal. The situation is even worse if multiple monitoring applications are active simultaneously, which is increasingly common as monitors are used for traffic engineering, SLA monitoring, intrusion detection, steering schedulers in GRID

computing, etc. Moreover, the processing requirements are increasing. Consider the following monitoring applications:

1. An intrusion detection system (IDS) checks the payload of every packet for worm signatures [31].
2. An application based on the 'Coralreef' suite keeps statistics for the ten most active flows [21].
3. A tool is interested in monitoring flows for which the port numbers are not known *a priori*. Such flows are found, for example, in peer-to-peer and H.323 multimedia flows where the control channels use well-known port numbers, while the data transfer takes place on dynamically assigned ports [32].
4. Multiple monitoring applications (e.g. `snort`, `tcpdump`, etc.) access identical or overlapping sets of packets.

In high-speed networks, none of these applications are catered to in the kernel in a satisfactory manner by existing solutions such as BPF, the BSD Packet Filter [25], and its Linux cousin, the Linux Socket Filter (LSF). In our view, they require a rethinking of the way packets are handled in the operating system.

In this paper, we discuss the implementation of the fairly fast packet filter (FFPF). FFPF introduces a novel packet processing architecture that provides a solution for filtering and classification at high speeds. FFPF has three ambitious goals: speed (high rates), scalability (in number of applications) and flexibility. Speed and scalability are achieved by performing complex processing either in the kernel or on a network processor, and by minimising copying and context switches. Flexibility is considered equally important, and for this reason, FFPF is explicitly extensible with native code and allows complex behaviour to be constructed from simple components in various ways.

On the one hand, FFPF is designed as an alternative to kernel packet filters such as CSPF [26], BPF [25], mmdump [32], and xPF [19]. All of these approaches rely on copying many packets to userspace for complex processing (such as scanning the packets for intrusion attempts). In contrast, FFPF permits processing at lower levels and may require as few as zero copies (depending on the configuration) while minimising context switches. On the other hand, the FFPF framework allows one to add support for any of the above approaches.

FFPF is not meant to compete with monitoring suites like Coralreef that operate at a higher level and provide libraries, applications and drivers to analyse data [21]. Also, unlike MPF [34], Pathfinder [3], DPF [17] and BPF+ [5], the goal of this research is not to optimise filter expressions. Indeed, the FFPF framework itself is language *neutral* and currently supports five different filter languages. One of these languages is BPF, and an implementation of `libpcap`¹ exists, which ensures not only that FFPF is backward compatible with many popular tools (e.g., `tcpdump`, `ntop`, `snort`, etc. [31]), but also that these tools get a significant performance boost (see Section 5). Better still, FFPF allows users to mix and match packet functions written in different languages.

To take full advantage of all features offered by FFPF, we implemented two languages from scratch: FPL-1 (FFPF Packet Language 1) and its successor, FPL-2. The main difference between the two is that FPL-1 runs in an interpreter, while FPL-2 code is compiled to fully optimised native code.

The aim of FFPF is to provide a complete, fast, and safe packet handling architecture that caters to all monitoring applications in existence today and provides extensibility for future applications. Since its first release in May 2003 we have constantly improved the code and gained a fair amount of experience in monitoring. We now feel that the architecture has stabilised and the ideas are applicable to systems other than FFPF as well. FFPF is available from `ffpf.sourceforge.net`. Some contributions of this paper are summarised below.

1. We generalise the concept of a ‘flow’ to a stream of packets that matches arbitrary user criteria.
2. Context switching and packet copying are reduced (up to ‘zero copy’).
3. We introduce the concept of a ‘flow group’, a group of applications that *share* a common packet buffer.
4. Complex processing is possible in the kernel or NIC (reducing the number of packets that must be sent up to userspace), while Unix-style filter ‘pipes’ allow for building complex flow graphs.

5. Persistent storage for flow-specific state (e.g., counters) is added, allowing filters to generate statistics, handle flows with dynamic ports, etc.

To our knowledge, few solutions exist that support *any* of these features and none that provide *all* in a single, intuitive, architecture. In this paper, we present the FFPF architecture and its implementation in the Linux kernel. The remainder of this paper is organised as follows. In Section 2, a high-level overview of the FFPF architecture is presented. In Section 3, implementation details are discussed. A separate section, Section 4 is devoted to the implementation of FFPF on the IXP1200. FFPF is evaluated in Section 5. Related work is discussed throughout the text and summarised in Section 6. Conclusions and future work are presented in Section 7.

2 FFPF high-level overview

The FFPF framework can be used in userspace, the kernel, the IXP1200 network processor, or a combination of the above. As network processors are not yet widely used, and (pure) userspace FFPF does not offer many speed advantages, the kernel version is currently the most popular. For this reason, we use FFPF-kernel to explain the architecture, and describe the userspace and network processor versions later. The main components are illustrated in Figure (1.a).

A key concept in FFPF is the notion of a *flow* which is different from what is traditionally thought of as a flow (e.g., a ‘TCP flow’). It may be thought of as a generalized socket: a flow is ‘created’ and ‘closed’ by an application and delivers a stream of packets, where the packets match arbitrary user criteria (e.g., “all UDP and TCP packets sent to port 554”, or “all UDP packets containing the CodeRed worm plus all TCP SYN packets”). The flow may also provide other application-specific information (e.g., traffic statistics).

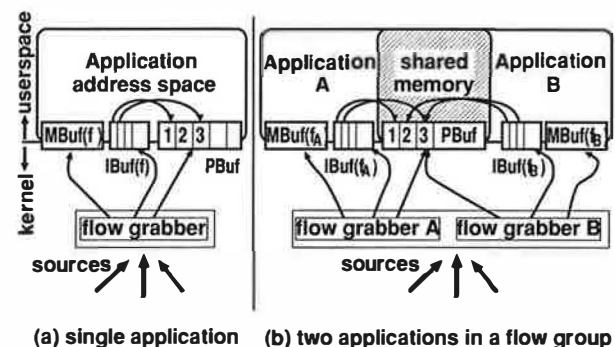


Figure 1: The FFPF architecture

A flow is captured by a *flow grabber*. For now, con-

¹<http://www.tcpdump.org/>

sider a flow grabber to be a filter that passes just the information (packets, statistics) in which the user is interested. Packets arrive in the system via one or more packet sources. Examples of packet sources include: (a) a network driver that interacts with a dumb NIC, (b) a smart NIC that interacts with FFPF directly, or (c) a higher-layer abstraction in the operating system that hides device-specific issues. A flow grabber receives the packets and if they correspond to its flow, stores them in a circular packet buffer known as *PBuf*. In addition, it places a pointer to this packet in a second circular buffer, known as the index buffer, or *IBuf*. Applications use the pointers in *IBuf* to find packets in *PBuf*.

The reason for using *two* buffers for capturing a flow is that while *IBuf* is specific to a flow, *PBuf* is *shared*. If the application opens two flows, there will be just one *PBuf* and two *IBufs*. If the flows are ‘overlapping’ (i.e., some packets in flow_a are also in flow_b), only one copy of each packet will be in *PBuf*. However, if a packet is in both flows, a pointer to it is placed in both *IBufs*. In other words, we do not copy packets to individual flows. Moreover, the buffers are memory mapped, so we do not copy between kernel and userspace either. We show later how *PBuf* can also be shared by multiple applications (as sketched in Figure (1.b)). Using memory mapping to avoid copying is a known technique, also used in monitoring solutions like DAG and SCAMPI [10, 30]. Edwards et al. also give userspace applications direct control over packet buffers, but provide an explicit API to access the buffers rather than memory mapping [15].

Thus far, we have assumed that a flow grabber is equivalent to a filter. In reality, a flow grabber can be a complex graph of interconnected filters, where a filter is defined as an element that takes a stream of packets as input and returns a (possibly empty) subset of this stream as output. In addition, a filter may provide arbitrary information about the traffic, e.g., statistics, intrusion alerts, etc. For this purpose, every filter has an associated *MBuf* (also memory mapped), which is a buffer that is used to produce results for applications, or to keep persistent state. It can also be used by the application to pass configuration parameters to the filter. For instance, in case of a ‘blacklist filter’ the application may store the addresses of the blacklist in *MBuf*. Note that the ability to perform more complex processing than just filtering, helps to reduce context switches, e.g., because applications that are interested in periodic statistics only and not in the packets themselves need not be scheduled for packet processing.

In later sections, we show that FFPF is language neutral, so that, for instance, BSD packet filters can be combined with filters written in other languages. In fact, the filters in a flow grabber are simple instantiations of fil-

ter *classes*, one of which may be the class of BPF filters. In addition to existing languages like BPF, we support two new languages (see Section 3.3) that are explicitly designed to exploit all features offered by FFPF. Among other things, they provide extensibility of the FFPF framework by their ability to call ‘external functions’ (provided these functions were previously registered with FFPF). External functions commonly contain highly optimised native or even hardware implementations of operations that are too expensive to execute in a ‘safe’ language (e.g., pattern matching, generating MD5 message digests).

We have covered most aspects of FFPF that are relevant if a single monitoring application is active. It is now time to consider what happens if multiple applications are present. For this purpose, we introduce a new concept, called the *flow group*. A flow group is a set of applications with the same *access rights* to packets, i.e., if one application is allowed to read a packet, all others in the same group may also access it. Flow groups are again used to minimise packet copying. Applications in the same group *share* a common *PBuf*. *PBuf* contains all packets for which one or more applications in the group have expressed interest. This is illustrated in Figure (1.b). If more than one group express interest in the packet, it is copied once per group, unlike existing approaches (such as BPF/LSF) which copy the packet to each application separately. This makes FFPF cheaper than other solutions when supporting multiple applications. In the current implementation, the flow group is determined by group id. In the future, we plan to provide applications with more explicit control over flow groups.

We see that FFPF demultiplexes packets to their respective flows *early*, i.e., well before they are processed by the kernel protocol stack. This is a tried technique that is also used in projects like LRP [14]. Unlike LRP, however, we do not place the packets themselves on application-specific queues, but only the corresponding pointers. Thus, it is possible to avoid copying both for demultiplexing purposes and for crossing the protection domain boundaries.

2.1 Receiving packets in a flow

An application may be interested in multiple flows. Flows are captured from a raw input stream in four steps. Firstly, a flow handle is created with the `flow_create()` operation. Creating a flow handle sets up a user-space data structure which is used as an identifier in all future operations on the flow, but does not result in any packets being captured. Secondly, the flow handle structure is *populated* using the `flow_populate()` operation by specifying for instance the graph of connected filters, callback functions and other parameters to be associated with the flow. The

result is a *flow definition* in user space consisting of a graph of filters that will capture the flow, associated callbacks, etc. Thirdly, the flow definition is used as blueprint to *instantiate* a ‘flow grabber’ which is done by calling the `flow_instantiate()` operation. Only at instantiation time are the filters that capture the flow instantiated and connected, provided the flow definition passes the *authorisation control* check (Section 3.4). Fourthly, an instantiated flow grabber by itself still does not capture packets; the flow grabber first needs to be *activated*. Conversely, an activated flow can be *paused* (and subsequently re-activated). Flow activation and pausing is performed using the `flow_activate()` and `flow_pause()` operations. Finally, a flow can be *closed* (`flow_close()`). When a flow is closed (or the corresponding application crashes), all flow state is destroyed. In the remainder of this paper, we will use the term ‘flow’ to refer both to the flow grabber (the code in the kernel that captures the flow), and to the packets captured by the flow grabber (the real ‘flow’), except where the distinction is important.

Instantiation is a separate step, because the flow specification is sent *in its entirety* to authorisation control, so that we can enforce that a packet function f (e.g., payload scanning) be allowed if and only if another function g (e.g., a filter passing only traffic from a specific subnet) is applied before (or after) f . Flow activation is also a separate step, as it gives administrators more accurate control over the start time (flow activation is more lightweight than flow instantiation).

2.2 Filter expressions

FFPF is language neutral, which means that different languages may be mixed. As mentioned earlier, we currently support five languages: BPF, FPL-1, FPL-2, C, and OKE-Cyclone. Support for C is limited to root users. The nature of the other languages will be discussed in more detail in Section 3. Presently, we only sketch how multiple languages are supported by the framework.

Figure (2.a) shows an example with two simplified flow definitions, for flows A and B , respectively. The grabber for flow A scans web traffic for the occurrence of a worm signature and saves the IP source and destination addresses of all infected packets. In case the signature was not encountered before, the packet is also handed to the application. Flow grabber B counts the number of fragments in web traffic. The first fragment of each fragmented packet is passed to the application.

There are a few things that we should notice. First, one of these applications is fairly complex, performing a full payload scan, while the other shows how state is kept regardless of whether a packet itself is sent to userspace. It is difficult to receive these flows efficiently using existing

packet filtering frameworks, because they either don’t allow complex processing in the kernel, or do not keep persistent state, or both. Second, both flows may end up grabbing the same packets. Third, the processing in both flows is partly overlapping: they both work on HTTP packets, which means that they first check whether the packets are TCP/IP with destination port 80 (first block in Figure 2). Fourth, as fragmentation is rare and few packets contain the CodeRed worm, in the common case there is no need for the monitoring application to get involved at all.

Figure (2.a) shows how these two flows can be accommodated. A common BPF filter selecting HTTP/TCP/IP packets is shared by both flows. They are connected to the flow-specific parts of the data paths. As shown in the figure, the data paths are made up of small components written in different languages. The constituent filters are connected in a fashion similar to UNIX pipes. Moreover, a pipe may be ‘split’ (i.e., sent to multiple other pipes, as shown in the figure) and multiple pipes may even be ‘joined’. Again, in UNIX fashion, the framework allows applications to create complex filter structures using simple components. A difference with UNIX pipes, however, is the method of connection: FFPF automatically recognises overlapping requests and merges the respective filters, thereby also taking care of all component interconnects.

Each filter has its own *IBuf*, and *MBuf*, and, once connected to a packet source, may be used as a ‘flow grabber’ in its own right (just like a stage in a UNIX pipe is itself an application). Filters may read the *MBuf* of other filters in their flow group (although we have not yet implemented synchronisation primitives to prevent races). In case the same *MBuf* needs to be written by multiple filters, the solution is to use function-like *filter calls* supported by FPL-1 and FPL-2, rather than pipe-like *filter concatenation* discussed so far. For filter call semantics, a filter is called *explicitly* as an external function by a statement in an FPL expression, rather than implicitly in a concatenated pipe. An explicit call will execute the target filter expression with the calling filter’s *IBuf* and *MBuf*. An example is shown in Figure (2.b), where a first filter call creates a hash table with counters for each TCP flow, while a second filter call scans the hash table for the top-10 most active flows. Both access the same memory area.

2.3 Construction of filter graphs by users

FFPF comes with a few constructs to build complex graphs out of individual filters. While the constructs can be used by means of a library, they are also supported by a simple command-line tool called `ffpf-flow`. For example, pronouncing the construct ‘ \rightarrow ’ as ‘connects to’

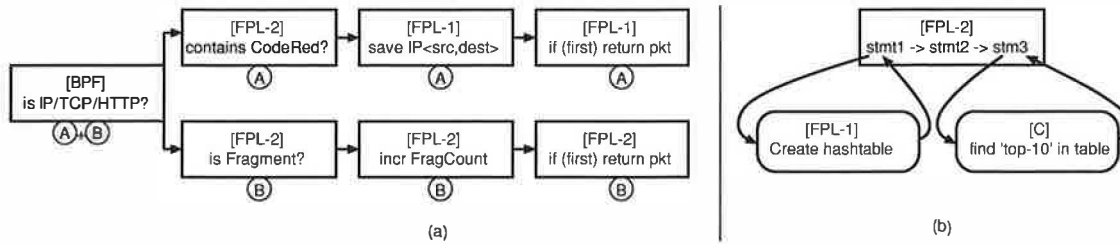


Figure 2: (a) combining different languages in two flows (A and B), (b) calling external functions from a single flow

and ' | ' as 'in parallel with', the command below captures two different flows:

```
./ffpf-flow \
  "(device,eth0) | (device,eth1) -> (sampler,2,4) -> \
    (FPL-2, "...") | (BPF, "...") -> (bytecount,,8)"
  "(device, eth0) -> (sampler,2,4) -> (BPF, "...") \
    -> (packetcount,,8)"
```

The top flow specification indicates that the grabber should capture packets from devices `eth0` and `eth1`, and pass them to a sampler that captures one in two packets and requires four bytes of `MBuf`. Next, sampled packets are sent both to an FPL-2 filter and to a BPF filter. These filters execute user-specified filter expressions (indicated by `'...'`), and in this example require no `MBuf`. All packets that pass these filters are sent to a bytecount 'filter' which stores the byte count statistic in in `MBuf` in an eight byte counter. The counter can be read directly from userspace, while the packets themselves are not passed to the monitoring application. The second flow has a prefix of two 'filters' in common with the first expression (devices are treated as filters in FFPF), but now the packets are forwarded to a different BPF filter, and from there to a packet counter.

As a by-product, FFPF generates a graphical representation of the entire filter-graph. A graph for the two flows above is shown in Figure 3. For illustration purposes, the graph shows few details. We just show (a) the configuration of the filter graph as instantiated by the users (the ovals at the top of the figure), (b) the filter instantiations to which each of the component filters corresponds (circles), and (c) the filter classes upon which each of the instantiations is based (squares). Note that there is only one instantiation of the sampler, even though it is used in two different flows. On the other hand, there are two instantiations of the BPF filter class. The reason is that the filter expressions in the two flows are different.

The ability to load and interconnect high-speed packet handlers in the kernel was also explored by Wallach et al., with an eye on integrating layer processing and reducing copying [33]. Similarly, Click allows programmers to load packet processing functions consisting of a configuration of simple elements that push (pull) data to (from) each other [28]. The same model was used in

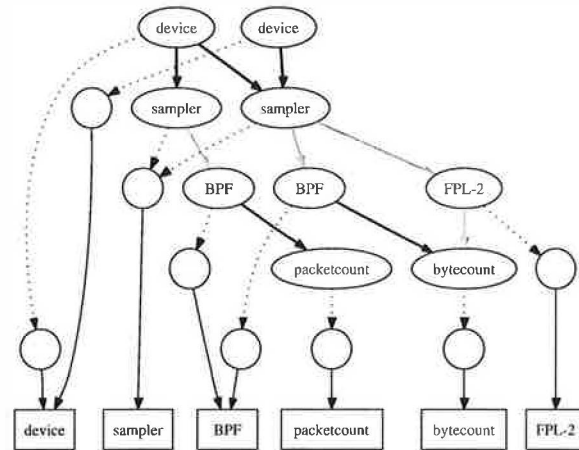


Figure 3: Auto-generated diagram of filter graph

the Corral, but with support for third parties that may add and remove elements at runtime [9]. The filter concatenation and support for a hierarchy that includes IXP1200s resembles paths in the Scout project [4]. Scout was not designed for monitoring *per se* and, hence, does not directly provide some of FFPF's features such as new languages or flow groups. Mono-lingual kernel-programming projects that also do not support these features include FLAME [1] and our own Open Kernel Environment [7] which provide high speed processing by loading native code (compiled Cyclone) in the kernel.

2.4 Processing

A key aspect to performance is that most processing takes place at the lowest possible level, e.g. in the kernel or network processor. For example, in the FFPF implementation on IXP1200 network processors, packet processing and buffer management are handled entirely by the IXP.

As shown in Figure 4, FFPF spans all three levels of the processing hierarchy: userspace, kernel, and network interface. Filters can be loaded at any of these levels.

The figure shows that filters from lower levels (e.g. the network card) may be connected to filters at higher levels. For instance, first-pass filtering may take place at the IXP1200, followed by more expensive processing at the host. A similar approach is found for instance in paths in the Scout OS [4]. Where in the processing hierarchy filters should be loaded depends on the availability of filter classes in each space and the trade-off in efficiency vs. stability at each level. Users need not concern themselves with this task as the deployment decision is made automatically by the FFPF toolkit.

The shaded areas in the figure indicate APIs that we developed on top of the native FFPF interface. The `libpcap` implementation guarantees backward compatibility with a host of applications. As shown in Section 5, running legacy `libpcap` applications on FFPF rather than on the existing Linux framework also leads to a significant performance boost. The MAPI is a very powerful monitoring API developed within the SCAMPI project [30]. Since FFPF's functionality exceeds that of both `pcap` and MAPI, the implementation of these interfaces involved just a few hours work. The FFPF toolkit supports automatic allocation of filters to the most optimal place in the processing hierarchy. Moreover, when a new flow grabber is instantiated, the toolkit automatically tries to merge it with already existing 'filter graphs', so that every common prefix is executed only once.

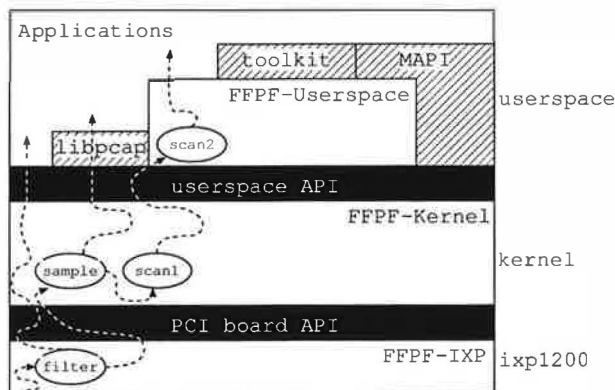


Figure 4: FFPF software structure (with some sample flows)

3 Implementation

3.1 The Buffers

Both *PBuf* and *IBuf* are circular buffers of N fixed size slots, with N a configurable constant. *PBuf* slots are large enough to hold maximum-size packets, while the slots in the index buffers hold two 32 bit values: an index in *PBuf* and the packet's 'classification result'

(the value returned by the filter). A packet is considered 'interesting' if the filter returns a non-zero result.

Applications read packets of a flow by indexing *PBuf* of filter f with the values in *IBuf*. By default, network packets are stored in *PBuf* from the link layer up. As applications access the index buffers, classification results are immediately available, normally within the same cache line. Although the indices *point* only to the packets in which they are interested, applications are able to see packets received by all others in the same flow group (but not those received by other groups).

3.1.1 Buffer management

Circular buffers in FFPF have two indices to indicate the current read and write positions. These are known as R and W , respectively. Whenever W catches up with R , the buffer is full. The way in which the system handles this case is defined by the buffer management system (BMS). The modular design of FFPF allows different BMSs to be used. The administrator chooses the BMS at startup time. The optimal choice depends on the mix of applications that will be executed and their relative importance. Currently, two BMSs have been defined. The first is known as 'slow reader preference' and is commonly used in existing systems. The second is known as 'fast reader preference' and is a novel way of ensuring that fast readers are not slowed down by tardy applications.

Slow reader preference. In SRP, as long as the buffer is full, all new packets are dropped. Both R and W are mapped read-only to an application's address space and updated in kernel or network card. The packet grabber in the kernel/card writes data in a group's *PBuf* and updates W until the buffer is full, i.e., until W catches up with the *slowest reader* in the group. Thus, the slowest reader in a group may block all other readers in that group. The R value of the slowest reader will be denoted by R^* . An application explicitly updates its own R by way of system call after it has processed a number of packets and, if needed, the kernel then also updates R^* . One of the keys to speed is that R need not be incremented by one for every packet that is processed. Instead, an application may process a thousand packets and then increment R by a thousand in one go. Doing so saves many kernel boundary crossings. A similar mechanism is used for DAG cards [10].

As an example, consider the implementation on IXP1200 network processors, where packet processing and buffer management is handled entirely by the IXP. The IXP receives a packet, places it in *PBuf*, updates W , receives the next packet, and so on. Meanwhile, the filters are executed in independent processing engines on

the network processor and determine whether a reference to the packet should be placed in the filters' index buffers. Assuming that the administrator chose to use 'zero-copy' packet handling (more about the various options in Section 4.3), applications access packets immediately, as the buffers are memory mapped through to userspace. While applications process the packets, the kernel is not used at all. Only after an application has processed n packets of a flow and decides to advance its R explicitly, the kernel is activated. On the reception of a request to advance an application's R , the kernel also calculates the new value of R^* and passes it to the packet receiving code on the IXP. In the extreme case, where a single application is active, the IXP code and application work fully independently and the number of interrupts and context switches is minimal. The way FFPF's SRP captures packets in a circular buffer and memory maps them to user space is similar to Luca Deri's PF_RING [13], although PF_RING copies packets to each application individually.

Fast reader preference. FRP is a departure from the 'traditional' way of dealing with buffer overflow. In FRP mode, FFPF keeps writing packets, regardless of the status of the readers, and it is the reader's responsibility to keep up. An application that fails to keep up may have older but still unread data overwritten by new packets. In this case, R is of no concern to FFPF and used only by the application. The idea is that applications check *after* they processed a set of packets, whether or not these packets were overwritten in the meantime (and hence whether the application should consider them lost after all). For this purpose, FFPF keeps a memory mapped *wrap counter* that is incremented each time W 'wraps' to zero. Using the counter, applications can check themselves whether the current value of W is greater than their value of R and thus whether the packets they just accessed were valid.

Suppose an application is about to access a set of 100 packets when the values of R , W and wrap counter are 50, 400, and 10, respectively. When the application has finished processing, it again checks these values and now finds that W is 450, while the wrap counter is 11. In other words, the writing process has wrapped and overwritten all packets that were just processed. The application will count these packets as dropped. Note that as a result the drop rate in a group may vary from application to application. It should be mentioned that FRP is not necessarily more efficient in terms of the total processing that is required for buffer management. Rather, it distributes this computation to the applications themselves, removing the dependencies between readers that exist in a centralised solution.

3.1.2 Filter-specific memory array

The third buffer in Figure 1 is the filter's memory array *MBuf*. It is used by both the filters in the kernel and the userspace application. User applications have read and write access to the memory arrays of their filters, so the arrays can be used to exchange data between the application and a filter expression. The *MBuf* area is *persistent*, i.e., its contents remain valid across multiple invocations of a filter. It is argued in [19] that the absence of persistent state is one of the major drawbacks of BPF. While [19] describes how BPF can be extended to also allow for persistent memory (and explicit switching between persistent and non-persistent memory is needed), this paper describes an approach in which it is part of the design from the outset.

A simple use case is a filter f which treats the entire memory array as a hash table that is used to count the number of packets received on all TCP/IP flows. The corresponding filter first checks whether a packet is TCP/IP. If so, it calculates a hash of the $\langle \text{ipsrc}, \text{ipdest}, \text{srcport}, \text{dstport} \rangle$ tuple and increments the counter stored at that location in the memory array. The result is that without intervention by the user application, the memory array contains the packet counts of all TCP/IP flows seen by the system (assuming the hash table is large enough). The implementation of this example is trivial if the language is capable of using persistent state. An example of such code in FPL-2 is shown in Figure 6 and will be discussed in Section 3.3.1.

3.2 The flows

Flows are captured by stringing together filters as explained in Section 2.2. The packets received in a flow can be read by the application in different ways. The simplest way is to read continuously from the buffer whenever a packet is available, e.g., using the `filter_getnext_pkt()` operation. Doing so, however, keeps the application polling constantly. From a CPU usage and context switching point of view, packets may be read more efficiently by blocking, e.g., until a certain number of packets has been received. FFPF offers two flavours of blocking: (a) `wait_for_n_pkts(n)`, a blocking call that only returns after n packets are received, and (b) installing a `filter_callback()` which is non-blocking itself and results in a callback of a registered callback function whenever n packets are received. At callback registration time, users specify how long the callback should remain active. Of course, even with `filter_getnext_pkt()` an application may block explicitly, e.g. by calling `sleep(10)` to process every 10 seconds all packets that were received in that period.

3.3 FFPF Packet Languages

While FFPF is language neutral, some languages are better suited to exploit the strengths of FFPF than others. BPF, for instance, can not by itself take advantage of FFPF's persistent state, extensibility, etc. For this reason, we developed two new languages for filter expressions, known as FPL-1 and FPL-2 (FFPF packet languages 1 and 2). They were designed to exploit all of FFPF's features. The main distinctions are that FPL-1 is a fairly slow interpreted stack language, while FPL-2 is fully optimised native code (based on registers), and that FPL-1 code can be self-modifying, while FPL-2 code is fixed. Also, the syntax in FPL-2 is much improved.

Given that FPL-1 has been around for a year now, why did we develop FPL-2? The reason is that although FPL-1 bytecode is fairly efficient, running it in an interpreter hurts performance. Moreover, as observed by McCanne and Van Jacobson: for modern processor architectures, stack-based languages are less efficient than register-based approaches [25]. For this reason, we developed a language that (1) compiles to fully optimised object code, and (2) is based on registers and memory, and (3) has a more readable syntax.

Apart from self-modification, there is little *functional* difference between FPL-1 and FPL-2. For this reason, we discuss 'FPL' as a general concept, using FPL-2 language constructs for illustration. A detailed explanation of FPL-1 and FPL-2 can be found in [8] and [12].

3.3.1 FPL

The FPL language is summarised in Figure 5. It supports all common integer types (signed and unsigned bits, nibbles, octets, words and double words) and allows expressions to get hold of any field in the packet header or payload in a friendly manner. Moreover, offsets in packets can be variable, i.e., determined by an expression. For convenience, an extensible set of macros allows use of shorthand for packet fields, e.g., instead of asking for bytes nine and ten to obtain the IP header's protocol field, a user may abbreviate to 'IP_PROTO'. We briefly explain constructs that are not intuitively clear.

FOR. The FOR loop construct is limited to loops with a pre-determined number of iterations. The break instruction, allows one to exit the loop 'early'. In this case (and also when the loop finishes), execution continues at the instruction following the ROF construct.

Registers and memory. FPL is able to access the filter's *MBuf* by means of the assignment operator. For instance, one may assign the content of a memory location to a register, perform a set of calculations, and then assign the value of the register back

operator-type	operator
Arithmetic	+, -, /, *, %, --, ++
Assignment	=, *=, /=, %=, +=, -= <=<, >>=, &=, ^=, =
Logical/Relational	==, !=, >, <, >=, <=, &&, , !
Bitwise	&, , ^, <<, >>
statement-type	operator
if/then/else	IF (expr) THEN stmt1 ELSE stmt2 FI
for()	FOR (initialise; test; update) stmts; BREAK; stmts; ROF
external function	EXTERN(filter, input, output)
hash()	INT HASH(start_byte,len,tablesize)
return a value	RETURN (val)
Data type	syntax
Register n^{\dagger}	R[n]
Memory location n	MEM[n]
Packets access:	
- byte $f(n)$	PKT.B[f(n)]
- word $f(n)$	PKT.W[f(n)]
- bit m in byte n	PKT.B[n].U1[m]
- byte m in word n	PKT.W[n].U8[m]
etc.	(many options, including macros)

Figure 5: FPL-2 language constructs ($^{\dagger}m$ and n arbitrary variables)

to memory. All accesses to *MBuf* are checked for bounds violations. An example of *MBuf* usage in FPL-2 is shown in Figure 6. The code implements the filter f mentioned in Section 3.1.2 that keeps track of how many packets were received on each TCP connection (assuming for simplicity that the hash is unique for each live TCP flow).

```
// count number of packets in every flow,
// by keeping counters in hash table
// (assume hash is unique for each flow)
IF (PKT.IP_PROTO == PROTO_TCP)
THEN
    // register = hash over TCP flow fields
    R[0] = Hash(14,12,256);
    // increment the pkt counter at this position
    MEM[ R[0] ]++;
FI
```

Figure 6: Example of FPL-2 code: count TCP flow activity

External functions. An important feature of FPL is extensibility and the concept of an 'external function' is key to extensibility, flexibility and speed. External functions are an explicit mechanism to introduce extended functionality to FFPF and add to flexibility by implementing the 'filter call' semantics shown in Figure (2.b). While they look like filters, the functions may implement anything that is considered useful (e.g., checksum calculation, pat-

tern matching). They can be written in any of the supported languages, but it is anticipated that they will often be used to call optimised native code performing computationally expensive operations.

In FPL, an external function is called using the `EXTERN` construct, where the parameters indicate the filter to call, the offset in `MBuf` where the filter can find its input data (if any), and the offset at which it should write its output, respectively. For instance, `EXTERN(f00, x, y)` will call external function `f00`, which will read its input from memory at offset `x`, and produce output, if any, at offset `y`. Note that FFPF does not prevent users from supplying bogus arguments. Protection comes from authorisation control discussed in Section 3.4 and from the compiler. The compiler checks the use of external functions in a filter. An external function's definition prescribes the size of the parameters, so whenever a user's filter tries to let the external function read its input from an offset that would make it stray beyond the bounds of the memory array, an error is generated. This is one of the advantages of having a 'trusted' compiler (see also Section 3.3.3. In addition, authorisation control can be used to grant users access only to a set of registered functions.

A small library of external filter functions has been implemented (including implementation of popular pattern matching algorithms, such as Aho-Corasick and Boyer-Moore). The implementation will be evaluated in Section 5. External functions in FPL can also be used to 'script together' filters from different approaches (e.g., BPF+ [5], DPF [17], PathFinder [3], etc.), much like a shell script in UNIX.

3.3.2 Monitoring application with dynamic ports

Many existing packet filters are not well suited for handling applications with dynamic ports. Such applications use control channels with well-known port numbers, while data transfer takes place over ports that are negotiated dynamically. Examples are found in peer-to-peer networks and multimedia streams that employ control protocols like RTSP, SIP and H.323 to negotiate port numbers for data transfer protocols such as RTP.

These flows are complex to monitor and the problem was considered important enough to develop a special-purpose tool (`mmdump`, not unlike `tcpdump`) to tackle it [32]. Like `xPF` [19], `mmdump` adds statefulness to the `pcap/BPF` architecture and in addition allows filters to be self-modifying. A filter may capture and inspect all control packets and if they contain the port number to be used for data, modify itself to also capture these packets.

While the same behaviour is supported in FFPF which allows an external function to extend the FPL-1 expression from which it was called (subject to authorization constraints), this may not be best way of handling the problem: self-modifying code is difficult to trace and debug. Moreover, there is a simpler way to monitor dynamic ports. For example, given that RTSP packets are sent on port 554, the filter in Figure 7 filters out all such packets and passes them to an external function `GetDynTCPDPortFromRTSP`. When called, the function scans all RTSP session packets for the occurrence of 'Transport', 'client_port' and 'server_port' to find the port numbers that will be used for data transfer (e.g., audio and video). These ports are stored in `MBuf` (lines 4-5). If the packet is not RTSP, we check if the destination port of the packets is in the array of port numbers and if so, return the value `TRUE` (lines 7-9), so that the packet is sent to userspace. In other words, *only* data packets of streams that are set up using RTSP are sent to userland. Note that the example is for illustration purposes only. It is a simplified version of what real applications would use. For instance, we only deal with transfers that use TCP (also for the data) and extract just a single destination port (while the traffic is likely to be bi-directional).

```

1. // R[0] initially 0 stores no. of dynports found
2. IF (PKT.IP_PROTO==PROTO_TCP) THEN
3.   IF (PKT.TCP_DPORT==554) THEN
4.     MEM[R[0]]=EXTERN("GetDynTCPDPortFromRTSP", 0, 0);
5.     R[0]++;
6.   ELSE
7.     FOR (R[1]=0; R[1] < R[0]; R[1]++)
8.       IF (PKT.TCP_DPORT == M[ R[1] ]) THEN
9.         RETURN TRUE;
10.    FI
11.  ROF
12. FI
12. RETURN FALSE;
```

Figure 7: Monitoring dynamic flows

3.3.3 Compile-time checks

The two FPL compilers are able to generate 'resource safe' code, i.e., it is possible to check *at compile time* how many resources can be consumed by an expression, how many loop iterations may be incurred, etc. Neither FPL language supports pointers and interaction with the rest of the kernel is limited to the explicitly registered external functions. Also, while it is not possible to determine the resource consumption of external functions *statically*, we are able to check (and control) *which* functions may be called from a filter. As a result, a simple authorisation check rejects filter expressions that do not agree with the local safety policy and no runtime checks for resource consumption are necessary. This is

explained in the next section. At runtime the code only checks for array bound violations, divide by zero, etc. By configuring the size of all buffers and slots as a power-of-2, bounds checking involves no more than a bitwise AND.

In an approach modelled after the OKE (see Section 3.5), the (trusted) FPL-2 compiler takes the filter expression, checks whether it is safe and if so, compiles it to a Linux kernel module which is subsequently compiled by `gcc`. It also generates a *compilation record*, which proves that this module was generated by the local (trusted) FPL-2 compiler. The proof contains the MD5 of the object code and is signed by the compiler (Figure 8). We check at load time whether the code was generated by a trusted compiler and whether the MD5 matches the code [8].

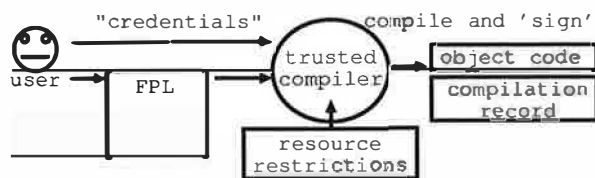


Figure 8: User compiles kernel module

3.4 Authorisation

It is important to note that the use of FFPF is not restricted to `root` users. Our view coincides with what was originally advocated in the `packetfilter` approach in Ultrix: limiting access to tools like `tcpdump` to a specific user (as found in many existing systems) is a design decision, not an axiom. Moreover, we think it is flawed. In FFPF, ordinary users may receive (in the form of credentials) explicit permission to monitor the network (possibly with restrictions, e.g., only packets with specific addresses).

For all control operations, e.g. when flow grabbers are instantiated or filters connected (Figure 9), users present authorisation information. The information required depends on language, user id and group id. When an FPL-2 filter is instantiated, users provide both object code and compilation record. The authorisation module checks whether the code is indeed FPL-2 code generated by the local compiler. If so (and all other authorisation checks are also passed), FFPF instantiates it. All authorisation information is normally stored in a single directory indicated by an environment variable. As a result, the checks are transparent to the user.

Authorisation control is implemented as a stand-alone daemon called at instantiation time. The daemon compares flow definitions both with the users' credentials and with the host's security policy and returns a verdict ('ac-

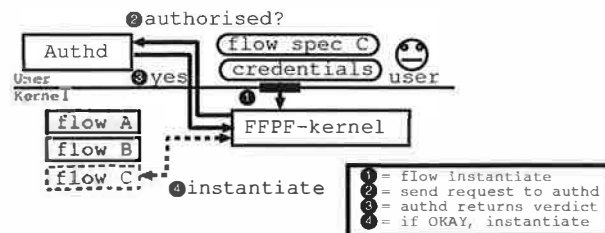


Figure 9: User loads module in the kernel

cept' or 'reject'). Credentials and trust management in FFPF are implemented in KeyNote [6]. The daemon provides fine-grained control that allows for complex policies. For instance, although we don't use most of them in the current distribution for simplicity reasons, it is possible to specify such policies as: (a) a call to external function `strsearch` is permitted for packets arriving on `eth0` *only* if it is preceded by a sampling function, (b) all calls to an external function `drop` *must* be followed by a return statement, (c) if no call is made to an external sampling function, the callback that is requested should wait for at least 1000 packets (e.g., to limit the number of callbacks), and (d) filter *x* may only be connected to filter *y*, etc. These policies can only be checked if the entire flow definition is available. The examples show that authorisation control guards against 'unsafe' flow grabbers, but can also be used to guard against 'silly' mistakes.

Authorisation control is optional. For instance, if the only party using FFPF is the system administrator, authorisation control may be left out to simplify management. A slightly modified version of the FFPF authorisation control daemon is also used in the SCAMPI network monitoring project [30].

3.5 Third-party external functions

The final two languages supported by FFPF are C and OKE-Cyclone. They are not intended to be used for packet processing by normal users on a day-to-day basis (although this is not precluded), but rather for implementing fast filters or external functions that can be called from FPL-1 or FPL-2. Writing kernel modules in C is too complex for most users, and writing code in the OKE is even more complex than that. We expect only power users to exploit the 'native code extensibility' features. Even so, once written and declared safe, the code and credentials needed to install such code can be given to third-parties.

External functions written in C and compiled as kernel modules can only be loaded by the system administrator. However, in the Open Kernel Environment [7]² it was shown how third-party users can load fully opti-

²Available from www.liacs.nl/~herbertb/projects/oke/

mised native code in the Linux kernel, without compromising safety in any way. OKE support was added to FFPF, so that even non-root users are allowed to load fast native functions in the kernel and register them with FFPF. Subsequently, these functions can be called by or connected to filter expressions just like ordinary external functions.

The FPL-2 way of injecting code was directly modelled after the OKE, so compilation and instantiation are as sketched in Figures 8 and 9, except that the language used in the OKE is *OKE-Cyclone*, a ‘crash-free’ version of C [20]. Unlike FPL-2, this is a language that supports pointer memory allocation and full interaction with the kernel. Accordingly, to be able to generate ‘resource safe’ code, the compiler must check and instrument the user code much more strictly. Depending on the credentials provided by the user the OKE compiler restricts the user code in terms of access to resources, e.g. CPU, heap, and stack usage, access to APIs, access to sensitive fields in packets, accesses to kernel heap, etc.

Using the OKE, users no longer depend on root users to load the desired functionality as native code. The cost of full resource control in the OKE is roughly 10% compared to plain C. Like authorisation control, the OKE is optional. We refer to [7] for a discussion of related work in safe kernel programming.

3.6 FFPF packet sources

Packets enter the FFPF framework via a call to an FFPF function called `hook_handle_packet()` which takes a packet as argument. As this is the only interface between the code responsible for packet capture and the FFPF packet handling module, it is easy to add new packet sources. Currently, three sources are implemented.

The first source, known as `netfilter`, captures packets from a netfilter hook. Netfilter is an efficient abstraction for packet processing in Linux kernels (from version 2.4 onward). The second source, known as `raw`, also works with older kernels. The third packet source, known as `ixp`, differs from the other two in that the IXP1200 device is assumed to be dedicated to monitoring in the FFPF framework³. As this packet source is a substantial project in and of itself, we will summarise its main characteristics in a separate section.

³If the IXP is used as a ‘normal’ NIC (e.g., as described in [23]), FFPF’s standard packet sources work without modification.

4 The IXP1200 packet source

4.1 The IXP1200 processor

The Intel IXP1200 runs at a clockrate of 232 MHz and is mounted on a Radisys ENP2506 board together with 8 MB of SRAM and 256 MB of SDRAM. The board contains two Gigabit network ports ①. Packet reception and packet transmission over these ports is handled by the code on the IXP1200 processor ②. The Radisys board is connected to a Linux PC via a PCI bus ③. The IXP itself consists of a StrongARM host processor running embedded Linux and six independent RISC processors, known as microengines. Each microengine has its own instruction store and register sets. On each of the microengines, registers are partitioned between 4 hardware contexts or ‘threads’ that have their own program counters and allow for zero-cycle context switches.

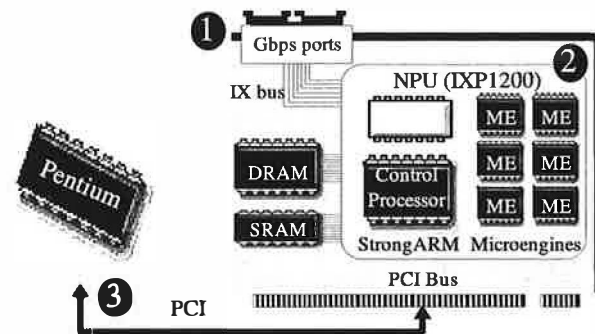


Figure 10: The IXP1200 NPU

4.2 FFPF on the IXP1200

The *PBuf* in the IXP implementation resides in SDRAM on the network card. FFPF maps the packet buffer as well as the other FFPF buffers into the host address space to support zero-copy functionality. A separate control structure, consisting of packet descriptors on a hardware-supported LIFO is kept in SRAM.

4.2.1 The microengines

A single microengine per Gigabit port is responsible for receiving and buffering packets in *PBuf*. All remaining microengines execute application-specific filter expressions. For this purpose, we implemented an FPL-2 compiler that generates Intel’s microengine-C. On each of the microengines a skeleton *main loop* with a slot for user code is provided by the FFPF framework. Users with the right credentials may ‘plug in’ FPL-2 expressions in this slot. When such a flow grabber is instantiated, the complete program is loaded on the microengine.

An FFPF IXP1200 filter is bound to a microengine's filter. As a consequence, the IXP1200 can support a maximum of five filters. As the IXP1200 is considered 'obsolete' and no longer supported by Intel, and newer versions of the IXP support more microengines at higher clockrates, both the number of filters that can be supported and their speeds may be expected to increase.

A filter uses all four threads to process the packets one by one. If the filter determines that the packet is interesting, the microengine places an index for the packet in the filter's *IBuf*. Otherwise, a special function `pkt_drop()` marks the packet as finished by setting a flag in the SRAM packet descriptor entry. In addition, it checks if all other filters are also finished with the packet. If so, the packet descriptor will be reclaimed.

4.2.2 StrongARM core components

The StrongARM components are responsible for control tasks, including initialization, loading and control of microengines, memory mapping of SDRAM to the host, initialization of different memory buffers, etc. It is also responsible for signalling across the PCI bus. For instance, in a 'slow-readers preference' implementation, the StrongARM receives the 'advance read pointer' messages from the host.

4.3 To copy or not to copy

While many research projects aim for zero-copy implementations, we argue that this is not always optimal. For this reason, we developed three implementations: (1) zero copy, (2) *always* copy packets to the host processor, and (3) only copy packets that have been marked as interesting by a host-side flow. Which version will be used by FFPF can be decided by the administrator at runtime.

The problem with zero-copy is that packet accesses from the host inevitably become slow, and if the average number of accesses per packet exceeds a certain threshold, the performance decreases. For instance, the zero-copy implementation just described works well, if most of the packet accesses are performed by the microengines and few or none by the host applications. Once the host application also needs to access the packet extensively, most reads have to cross the PCI bus. While some benefit may be expected from prefetching (reducing the overhead to less than a round-trip time), the penalty is still severe. If on the other hand, we had chosen the inverse zero-copy solution, whereby packets were immediately written to host memory and not stored in on-board SDRAM (ignoring potential bus bandwidth problems), host accesses would be optimised at the expense of the code on the microengines. We conclude that in situations where both the host and the IXP have an average number of accesses per packet that is substantial compared

to a single copy across the bus, copying the interesting packets once is always better than a zero-copy solution.

5 Experimental analysis

The FFPF architecture is arguably more complex than many of its competitors. A possible consequence of increasing expressiveness may be a decrease in performance of simple tasks. To verify FFPF's applicability in the general case we have directly compared it with the widely used Linux socket filter (LSF), by running identical queries through (a) libpcap with Linux' LSF back-end, and (b) libpcap based on an FFPF implementation. We realise that for various aspects of filtering faster solutions may exist, but since the number of different approaches is huge and none would be 'obvious' candidates for comparison, we limit ourselves to the most well-known competitor and compare under equivalent configurations (using the same BPF interpreter, buffer settings, etc.).

To show their relative efficiency we compare the two packet filters' CPU utilization (system load) using OProfile⁴. Since packet filtering takes place at various stages in kernel and userspace, a global measure such as system load can convey overall processing costs better than individual cyclecounters. Results of subprocesses are discussed using clockcycle counts later on. Both platforms have been tested with the same front-end, `tcpdump` (3.8.3). Use of the BPF interpreter was minimized as much as possible: only a return statement was executed. All tests were conducted on a 1.2 GHz Intel P3 workstation with a 64/66 PCI bus running Linux 2.6.2 with the new network API (NAPI), using FFPF with FRP and circular buffers of 1000 slots.

5.1 Packet sniffing performance

Figure 11 shows the overhead incurred by running the packet filters at increasing bitrates for 1500 byte packets (600Mbps is the maximum rate we are able to generate reliably from a single source). While not shown in the figure, we verified that packet size plays no role in this experiment, only the packet rate. In general, we can see that FFPF makes more efficient use of the system than LSF, as the amount of FFPF idle time for high rates may exceed that of LSF by a factor of two, depending on the capture length. Unlike LSF, FFPF performance, while always better than LSF for high rates, depends strongly on the maximum packet capture length. Varying the number of slots in the packet buffer has a similar effect on performance, which leads us to conclude that it is probably caused by memory access and cache behaviour. As LSF lacks a circular buffer, cache misses will be rare.

⁴<http://oprofile.sourceforge.net>

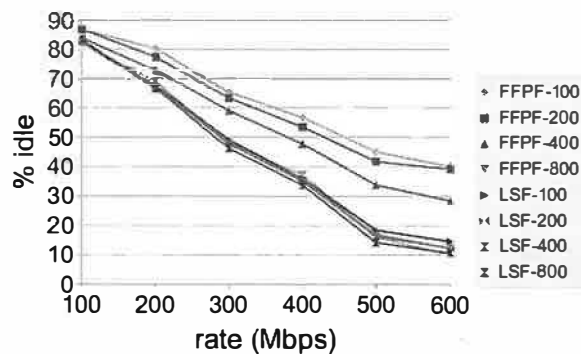


Figure 11: System idle time for FFPF and LSF as a function of the bandwidth for different capture lengths

Larger caches, or tweaking of buffer size helps to alleviate the dependency. However, this was not done in these experiments. The drop rate for FFPF in all of these configurations is negligible, while for LSF at 600 Mbps the drop rate is 2-3%, depending on the capture size.

The use of shared buffers in FFPF reduces copying and context switching, especially if the number of applications increases. It is our hypothesis that network monitoring will be increasingly important and that multiple different applications will want to filter overlapping traffic (e.g., for intrusion detection, traffic engineering, a sysadmin interested in an overview of activity of protocols, etc.).

Figure 12 shows, for high bitrate, how the two frameworks scale when starting an increasing number of `tcpdump` applications with overlapping flows. Since LSF duplicates much of the work for each application, it quickly saturates. We should point out that for reasons unknown to us, `OProfile` never reports 0% idle time (the minimum is always 2-3%). Even with just two simultaneous applications LSF reaches maximum system load and consequently starts dropping packets. With 6 client applications LSF drops between 64% (LSF-100) and 75% (LSF-800) of all incoming packets. FFPF, on the other hand, drops 10% (FFPF-100) to 15% (FFPF-800).

Interestingly, as the CPU load never reaches 100% for FFPF, the drop cannot be attributed to starvation. Rather it is caused by buffer overflow: by keeping the number of *PBuf* slots constant throughout the experiments (1000 slots), there were not enough slots to support six parallel client applications. Increasing the number of slots will decrease the droprate due to buffer overflow, but will increase overall system load due to cache and line misses. However, even without tweaking FFPF clearly outperforms LSF. Its more *gradual* performance degradation can be expected as little work is duplicated. Filtering is handled in the kernel and duplicate tasks are merged.

The remaining performance penalty is therefore related only to userspace data output and the remaining context switches.

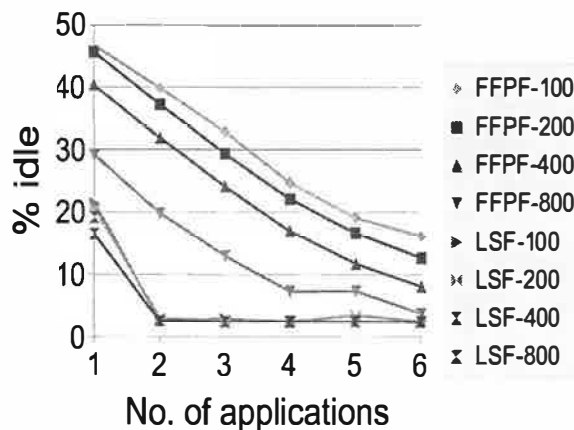


Figure 12: Idle time as function of the no. of concurrent applications for various capture lengths at 600Mbps

5.2 Analysis of operational costs

We have shown that FFPF increases packet filtering efficiency even for relatively simple tasks. The previous tests fail to show, however, where the performance gains originate and how the system would operate with more complex filters. Table 1 breaks down the overhead in several subtasks.

Rows 1 – 4 deal with general overhead, namely the calling of a filter, the total overhead per filter in the flowgraph (with filters that return immediately after being called to show only framework overhead), the saving of an element in an index buffer and the saving of a 1500B packet to *PBuf*. The decrease in cost by a factor 50 for saving a reference in *IBuf* over saving a full packet shows that in the presence of overlapping flows, FFPF's flowgroups can truly increase efficiency. This, combined with memory mapping of buffers, is perhaps the most important factor to the gradual degradation of performance when running multiple applications.

Rows 5 – 8 show resource consumption for a number of often executed filters, namely the Aho-Corasick pattern matching algorithm used in `snort` [31], and a simple `tcpdump` filter⁵ executed in `FPL2` code and `BPF` respectively. Rows 7 and 8 show that `FPL2` is four times as efficient as `BPF`, even for such a trivial filter. While not shown, cost savings grow with expression complexity (as expected). Unfortunately, the performance of really elaborate filters, such as those shown in Figures 6

⁵`!p src 192.168.1.3 and ip proto \udp and dst port 54321"`

	task	cycles
1	calling a filter	71
2	single filter stage in <i>fwgrabber</i>	171
3	saving index in <i>IBuf</i>	154
4	storing packet in <i>PBuf</i>	7479
5	waking up user process	624
6	snort's Aho-Corasick algorithm (match)	1000
7	same but without match	9900
8	FPL-2 filter	185
9	BPF filter	740

Table 1: Breakdown of various types of overhead in cycles

and 7, cannot be compared, as such complex filters cannot be expressed in BPF.

Pattern matching can also be seen to be costly. We show the case where an application (e.g., *snort*) is only interested in packets that contain a signature. Especially when a signature is *not* found after scanning the entire packet processing costs are high (the result shown is for 1500 byte packets). By executing this function in the kernel, FFPF eliminates a journey to userspace for *every* packet, avoiding unnecessary packet copies, context switches and signalling. Note that even compared to the high overhead of pattern matching, the overhead of storing packets is significant.

The complete cost of context switching is hard to measure due largely to the asynchronous nature of userspace/kernel communication. One measure that is quantifiable is the cost to wake up a user process, row 5 in Table 1. At 600 cycles (4 times the overhead of a filter stage), this is a significant cost. To minimize this overhead users can reduce communication by batching packets. Waking up a client process only once every N packets reduces this type of overhead by $N - 1$. In FFPF, N is configured by the size of the circular buffers and can be thousands of packets.

Furthermore, comparing filtering (row 5 – 8) and framework (rows 1 – 4) overhead shows that costs due to FFPF's complexity contributes only a moderate amount to overall processing. Finally, we discuss in a related publication that the IXP implementation is able to sustain full Gigabit rates for the same simple filter that was used for Figure 1, while a few hundred Mbps can still be sustained for complex filters that check every byte in the packet [29]. As the FPL-2 code on the IXP is used as pre-filtering stage, we are able to support line rates without being hampered by bottlenecks such as the PCI bus and host memory latency, which is not true for most existing approaches. We conclude that FFPF can be used as an efficient solution for both simple (e.g. BPF) and more complex (sampling, pattern matching) tasks.

6 Related work

Much of the related work was discussed in the text. In this section, we discuss projects that, although related, could not easily be linked with any *specific* aspect of FFPF.

MPF enhances the BPF virtual machine with new instructions for demultiplexing to multiple applications and merges filters that have the same prefix [34]. This approach is generalised by PathFinder which represents different filters as predicates of which common prefixes are removed [3]. PathFinder is interesting in that it is amenable to implementation in hardware. DPF extends the PathFinder model by introducing dynamic code generation [17]. BPF+ [5] shows how an intermediate static single assignment representation of BPF can be optimised, and how just-in-time-compilation can be used to produce efficient native filtering code. All of these approaches target filter optimisation especially in the presence of many filters, and as a result are not supported directly in FFPF (although it is simple to add them as external functions). With FPL-2, FFPF relies on gcc's optimisation techniques and on external functions for expensive operations.

Like FPL-2, and DPF, the Windmill protocol filters also target high-performance by compiling filters in native code [24]. And like MPF, Windmill explicitly supports multiple applications with overlapping filters. However, compared to FPL-2, Windmill filters are fairly simple conjunctions of header field predicates. MPF extends the BPF instruction set to exploit the fact that most filters concern the same protocol, so that common filter tests can be collapsed. It seems that the support is at the level of assembly instructions which makes it fairly hard to use. Moreover, for each of these approaches packets are still *copied* to individual processes and require a context switch to perform processing other than filtering. As FFPF is extensible and language neutral, each of these approaches can be added to FFPF if needed.

Operating systems like Exokernel, and Nemesis [16, 22] allow users to add code to the operating system and implement single address spaces to minimise copying. While FFPF no doubt can be efficiently implemented on these systems, one of its strengths is that it minimises copying on a very popular OS that does not have a single address space.

Support for high-speed traffic capture is provided by OCxMon [2]. Like the work conducted at Sprint [18], OCxMon supports DAG cards to cater to multi-gigabit speeds [10]. Unlike FFPF, both approaches have made an *a priori* decision not to capture the entire packet at high speeds.

Nprobe is aimed at monitoring multiple protocols [27] and is therefore, like Windmill, geared towards applying protocol stacks. Also, Nprobe focuses on disk bandwidth

limitations and for this reason captures as few bytes of the packets as possible. FFPF has no *a priori* notion of protocol stacks and supports full packet processing.

Gigascope is a stream database for network analysis that resembles FFPF in that it supports an SQL-like stream query language that is compiled and distributed over a processing hierarchy which may include the NIC itself [11]. The focus is on data management and there is no support for backward compatibility, persistent storage or handling dynamic ports.

Most related to FFPF is the SCAMPI architecture which also pushes processing to the lowest levels [30]. SCAMPI borrows heavily from the way packets are handled by DAG cards [10]. It assumes the hardware can write packets immediately in the applications' address spaces and implements access to the packet buffers through a userspace daemon. Common NICs are supported by standard `pcap` whereby packets are first pushed to userspace. Moreover, SCAMPI does not support user-provided external functions, supports a single BMS and relies on traditional filtering languages (BPF). Finally, SCAMPI allows only a non-branching (linear) list of functions to be applied to a stream.

7 Conclusions and future work

In this paper, we discussed the architecture and implementation of the fairly fast packet filter. FFPF provides a complete monitoring platform that caters to most applications. It was shown to be both more flexible and more efficient than existing approaches. Speed is gained by minimising both packet copying and context switching, pushing processing to the lowest levels, and executing computationally expensive functions as native code. It was demonstrated that FFPF outperforms Linux socket filters even for traditional applications that make no use of FFPF's more advanced features. The concepts of flows and flow groups, the concatenation of expressions, the buffering mechanism that favours fast flows and the minimisation of copying are generic mechanisms that may serve as the basis for fast packet processing in any OS.

In a future version of FFPF, we will explore the notion of flow groups further. Specifically, readers that are 'too slow' will be automatically placed in a separate flow group, lest they hinder fast applications. Also, we will shortly release a version of the architecture for use in application domains other than monitoring. For instance, in addition to packet reception, this version will be able to block, edit and (re)transmit packets, allowing for uses such as firewalling, network address translation and routing. Most of the required functionality is implemented, but currently not enabled in FFPF. Finally, we are in the process of developing a distributed version of FFPF.

Acknowledgments

This work was partly supported by the EU SCAMPI project IST-2001-32404, while Intel provided the IXP1200 network cards. A massive thanks is owed to the following people for commenting on earlier versions of this paper: Luca Deri (Netikos), Kobus van der Merwe (AT&T Labs), Andrew Moore (Cambridge University, UK), Sean Rooney (IBM Research, Zurich), and Jeffrey Mogul (HP Labs).

References

- [1] Kostas G. Anagnostakis, S. Ioannidis, S. Miltchev, and Michael B. Greenwald. Open packet monitoring on flame: Safety, performance and applications. In *Proc. of IWAN'02*, Zuerich, Switzerland, December 2002.
- [2] J. Apisdorf, k claffy, K. Thompson, and R. Wilder. Oc3mon: Flexible, affordable, high performance statistics collection. In *1996 USENIX LISA X Conference*, pages 97–112, Chicago, IL, September 1996.
- [3] Mary L. Bailey, Burra Gopal, Michael A. Pagels, Larry L. Peterson, and Prasenjit Sarkar. Pathfinder: A pattern-based packet classifier. In *Operating Systems Design and Implementation*, pages 115–123, 1994.
- [4] A. Bavier, T. Voigt, Wawrzoniak M, L. Peterson, and P. Gunningberg. Silk: Scout paths in the linux kernel, tr 2002-009. Technical report, Dept. of Information Technology, Uppsala University, Uppsala, Sweden, February 2002.
- [5] A. Begel, S. McCanne, and S. L. Graham. BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture. In *SIGCOMM*, pages 123–134, 1999.
- [6] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. The KeyNote trust-management system version 2. *Network Working Group, RFC 2704*, September 1999.
- [7] H. Bos and B. Samwel. Safe kernel programming in the OKE. In *Proc. of OPENARCH'02*, New York, USA, June 2002.
- [8] Herbert Bos and Georgios Portokalidis. Packet monitoring at high speed with FFPF. Technical Report TR-2004-01, LIACS, Leiden University, Leiden Netherlands, January 2004.
- [9] Herbert Bos and Bart Samwel. The OKE Corral: Code organisation and reconfiguration at runtime using active linking. In *Proceedings of IWAN' 2002*, Zuerich, Sw., December 2002.
- [10] J. Cleary, S. Donnelly, I. Graham, A. McGregor, and M. Pearson. Design principles for accurate passive measurement. In *Proceedings of PAM*, Hamilton, New Zealand, April 2000.
- [11] Chuck Cranor, Theodore Johnson, Oliver Spatschek, and Vladislav Shkapenyuk. Gigascope: a stream database for network applications. In *Proceedings of the 2003 ACM*

- SIGMOD international conference on on Management of data*, pages 647–651. ACM Press, 2003.
- [12] Mihai Cristea and Herbert Bos. A compiler for packet filters. In *Proc. of ASCI*, Netherlands, June 2004.
 - [13] Luca Deri. Improving passive packet capture: beyond device polling. <http://luca.ntop.org/>, 2004.
 - [14] Peter Druschel and Gaurav Banga. Lazy receiver processing (lrp): a network subsystem architecture for server systems. In *Proceedings of the second USENIX symposium on Operating systems design and implementation*, pages 261–275, 1996.
 - [15] Aled Edwards, Greg Watson, John Lumley, David Banks, Costas Calamvokis, and C. Dalton. User-space protocols deliver high performance to applications on a low-cost gb/s lan. *SIGCOMM Comput. Commun. Rev.*, 24(4):14–23, 1994.
 - [16] D. R. Engler, M.F. Kaashoek, and J.W. O'Toole Jr. The exokernel approach to extensibility (panel statement). In *Proc. of OSDI'94*, page 198, Monterey, Ca., November 1994.
 - [17] Dawson R. Engler and M. Frans Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *SIGCOMM'96*, pages 53–59, 1996.
 - [18] G. Iannaccone, C. Diot, I. Graham, and N. McKeown. Monitoring very high speed links. In *ACM SIGCOMM Internet Measurement Workshop 2001*, September 2001.
 - [19] S. Ioannidis, K. G. Anagnostakis, J. Ioannidis, and A. D. Keromytis. xPF: packet filtering for low-cost network monitoring. In *Proc. of HPSR'02*, pages 121–126, May 2002.
 - [20] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX 2002 Annual Technical Conference*, June 2002.
 - [21] Ken Keys, David Moore, Ryan Koga, Edouard Lagache, Michael Tesch, and k claffy. The architecture of Coral-Reef: an Internet traffic monitoring software suite. In *PAM2001*. CAIDA, RIPE NCC, April 2001.
 - [22] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *JSAC*, 14(7), September 1996.
 - [23] Kenneth Mackenzie, Weidong Shi, Austen McDonald, and Ivan Ganey. An intel IXP1200-based network interface. <http://www.ixp1200.com/resources/>, 2003.
 - [24] G. Robert Malan and F. Jahanian. An extensible probe architecture for network protocol performance measurement. In *Computer Communication Review, ACM SIGCOMM, volume 28, number 4*, Vancouver, Canada, October 1998.
 - [25] S. McCanne and V. Jacobson. The BSD Packet Filter: A new architecture for user-level packet capture. In *Proc. 1993 Winter USENIX conference*, San Diego, Ca., January 1993.
 - [26] J.C. Mogul, R.F. Rashid, and M.J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proc. of SOSR'87*, pages 39–51, Austin, Tx., November 1987. ACM.
 - [27] A. Moore, J. Hall, C. Kreibich, E. Harris, and I. Pratt. Architecture of a network monitor. in *proc. of PAM'03*, 2003.
 - [28] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The Click modular router. In *Symposium on Operating Systems Principles*, pages 217–231, 1999.
 - [29] Trung Nguyen, Willem de Bruijn, Mihai Cristea, and Herbert Bos. Scalable network monitors for high-speed links: a bottom-up approach. In *IPOM'04*, Beijing, China, 2004.
 - [30] M. Polychronakis, E. Markatos, K. Anagnostakis, and A. Oslebo. Design of an application programming interface for ip network monitoring. In *Proc. of NOMS'02*, Seoul, Korea, April 2004.
 - [31] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 1999 USENIX LISA Systems Administration Conference*, 1999.
 - [32] J. van der Merwe, R. Caceres, Y. Chu, and C. Sreenan. Mmdump - a tool for monitoring internet multimedia traffic. *ACM Computer Communication Review*, 30(4), October 2000.
 - [33] D. Wallach, D. Engler, and M.F. Kaashoek. ASHs: Application-specific handlers for high-performance messaging. In *Proc. of SIGCOMM'96*, pages 40–52. ACM Press, 1996.
 - [34] M. Yuhara, B.N. Bershad, C. Maeda, and J. E. B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *USENIX Winter*, pages 153–165, 1994.

Energy-Efficiency and Storage Flexibility in the Blue File System

Edmund B. Nightingale and Jason Flinn

*Department of Electrical Engineering and Computer Science
University of Michigan*

Abstract

A fundamental vision driving pervasive computing research is access to personal and shared data anywhere at anytime. In many ways, this vision is close to being realized. Wireless networks such as 802.11 offer connectivity to small, mobile devices. Portable storage, such as mobile disks and USB keychains, let users carry several gigabytes of data in their pockets. Yet, at least three substantial barriers to pervasive data access remain. First, power-hungry network and storage devices tax the limited battery capacity of mobile computers. Second, the danger of viewing stale data or making inconsistent updates grows as objects are replicated across more computers and portable storage devices. Third, mobile data access performance can suffer due to variable storage access times caused by dynamic power management, mobility, and use of heterogeneous storage devices. To overcome these barriers, we have built a new distributed file system called BlueFS. Compared to the Coda file system, BlueFS reduces file system energy usage by up to 55% and provides up to 3 times faster access to data replicated on portable storage.

1 Introduction

Storage technology for mobile computers is evolving rapidly. New types of storage such as flash memory offer performance and energy characteristics that differ substantially from those of disk drives. Further, the limited battery lifetime of mobile computers has made power management a critical factor in the design of mobile I/O devices. Finally, a dramatic improvement in storage capacity is enabling users to carry gigabytes of data in a pocket-sized device. Collectively, these changes are causing several of the assumptions made in the design of previous mobile file systems to no longer hold.

In this paper, we present the Blue File System (BlueFS), a distributed file system for mobile computing that addresses the new opportunities and challenges created by the evolution of mobile storage. Rather than embed static assumptions about the performance and availability of storage devices, BlueFS has a *flexible cache hi-*

erarchy that adaptively decides when and where to access data based upon the performance and energy characteristics of each available device. BlueFS's flexible cache hierarchy has three significant benefits: it extends battery lifetime through energy-efficient data access, it lets BlueFS efficiently support portable storage, and it improves performance by leveraging the unique characteristics of heterogeneous storage devices.

BlueFS uses a *read from any, write to many* strategy. A kernel module redirects file system calls to a user-level daemon, called Wolverine, that decides when and where to access data. For calls that read data, Wolverine orders attached devices by the anticipated performance and energy cost of fetching data, then attempts to read information from the least costly device. This lets BlueFS adapt when portable storage devices are inserted or removed, when the dynamic power management status of a device changes, or when wireless network conditions fluctuate.

For calls that write data, Wolverine asynchronously replicates modifications to all devices attached to a mobile computer. Wolverine aggregates modifications in per-device write queues that are periodically flushed to storage devices. Asynchrony hides the performance cost of writing to several devices, while aggregation saves energy by amortizing expensive power state transitions across multiple modifications.

We have implemented BlueFS as a Linux file system that runs on both laptops and handhelds. Our results show that BlueFS reduces file system energy usage by up to 55% and interactive delay by up to 76% compared to the Coda distributed file system. BlueFS also provides up to 3 times faster access to data replicated on portable storage. Further, BlueFS masks the performance impact of disk power management and leverages new types of storage technology such as flash memory to save power and improve performance.

We begin in the next section with a discussion of related work. Sections 3 and 4 outline BlueFS's design goals and implementation. We present an evaluation of BlueFS in Section 5 and discuss our conclusions in Section 6.

2 Related work

BlueFS distinguishes itself from prior distributed file systems by using an adaptive cache hierarchy to reduce energy usage and efficiently integrate portable storage.

Energy management has not been a primary design consideration for previous distributed file systems; however, several recent projects target energy reduction in the domain of local file systems. Researchers have noted the energy benefits of burstiness for local file systems [6, 15] and have provided interfaces that enable applications to create burstier access patterns [24]. Rather than defer this work to applications, BlueFS automatically creates burstiness by shaping device access patterns. BlueFS further reduces client energy usage by dynamically monitoring device power states and fetching data from the device that will use the least energy. Finally, BlueFS proactively triggers power mode transitions that lead to improved performance and energy savings.

Several ideas that improve file system performance may also lead to reduced energy usage. These include the use of content-addressable storage in systems such as LBFS [12] and CASPER [21], as well as memory/disk hybrid file systems such as Conquest [23]. We believe that these ideas are orthogonal to our current design.

Previous file systems differ substantially in their choice of cache hierarchy. NFS [13] uses only the kernel page cache and the file server. AFS [8] adds a single local disk cache — however, it always tries to read data from the disk before going to the server. xFS [3] employs cooperative caching to leverage the memory resources of networked workstations as a global file cache. Each static hierarchy works well in the target environment envisioned by the system designers precisely because each environment does not exhibit the degree of variability in device access time seen by mobile computers. In contrast, BlueFS's use of an adaptive hierarchy is driven by the expectation of a highly dynamic environment in which access times change by orders of magnitude due to dynamic power management, network variability, and devices with different characteristics.

The file system most similar to BlueFS is Coda [9], which is also designed for mobile computing. Like AFS, Coda uses a single disk cache on the client — this cache is always accessed in preference to the server. BlueFS borrows several techniques from Coda including support for disconnected operation and asynchronous reintegration of modifications to the file server [11].

Unlike BlueFS, Coda has not made energy-efficiency a design priority. Although Coda did not originally support portable storage, recent work by Tolia et al. [20] has used lookaside caching to provide limited, read-only support for portable devices. Lookaside caching indexes

files on a portable device by their SHA-1 hash. If Coda does not find a file in its local disk cache, it retrieves the attributes and SHA-1 hash for that file from the server. If a file with the same hash value exists in the index, Coda reads the file from portable storage.

Tolia's work examines how one can best support portable storage while making only minimal changes to an existing file system and its usage model. In contrast, our work integrates support for portable storage into the file system from the start. Our clean-sheet approach yields two substantial benefits. First, BlueFS support is not read-only. BlueFS keeps each portable device up to date with the latest files in a user's working set and with the latest version of files that have affinity for the device. Since all modifications written to a portable device are also written to the file server, BlueFS does not lose data if a portable device is misplaced or stolen. Second, BlueFS provides substantial performance improvements by maintaining consistency at the granularity of storage devices rather than clients. This lets BlueFS access files on portable storage devices without fetching attributes or hashes from the server.

PersonalRAID [17] uses portable storage to synchronize disconnected computers owned by the same user. The target environment differs from that of a distributed file system in that data is only shared between users and computers through synchronization with the portable device. Also, one must have the portable device in order to access data on any computer. Since PersonalRAID does not target mobile computers explicitly, it does not make energy-efficiency a design goal.

Bayou [19], Segank [18], and Footloose [14] take a peer-to-peer approach to mobile computing. These systems allow cooperating computers to share data through pairwise exchange. Many of the principles employed in BlueFS such as adaptive caching to multiple devices and tight integration with power management could also be applied in such peer-to-peer systems.

3 Design goals

In this section, we outline the three primary design goals of the Blue File System. We also describe how BlueFS addresses each goal.

3.1 Change is constant

Our first design goal is to have BlueFS automatically monitor and adapt to the performance and energy characteristics of local, portable, and network storage. Mobile storage performance can vary substantially due to:

- *deployment of new types of storage devices.* In recent years, some handheld computers have used flash or battery-backed RAM for primary

storage. Compared to disk storage, flash offers excellent performance for read operations but poor performance for small writes [25]. Battery-backed DRAM offers excellent performance for reads and writes, but can sacrifice reliability. Finally, the performance of new portable devices such as mobile disks often lags far behind that of their fixed counterparts [26].

- *the impact of power management.* Storage and network devices use aggressive power management to extend the battery lifetime of mobile computers. However, the performance impact of power management is often substantial. Consider disk drives that cease rotation during idle periods to save energy [5]. The next access is delayed several hundred milliseconds or more while the disk spins up. Similarly, when network interfaces are disabled or placed in power saving modes, the latency to access remote servers increases greatly [1]. The end result is that storage access latency can change by several orders of magnitude when a device transitions to a new power mode.
- *network variability.* Access times for remote storage are affected by fluctuations in network performance due to mobility. For example, when latency to a network server is low, the best performance will often be achieved by eschewing local storage and fetching data directly from the server. When latency is high, local or portable storage is best.

BlueFS adapts to storage variability by passively monitoring the performance of each local, portable, and remote storage option available to the mobile computer. It also monitors the power state of each device. When reading data, BlueFS orders available storage options by estimated performance and energy cost. It first tries to obtain needed data from the lowest cost device. If that option fails, it tries the next device. To mitigate the impact of searching devices that do not contain the desired data, it maintains an *enode cache* that records which devices hold recently used files.

On the write path, BlueFS places modifications in per-device operation logs and asynchronously writes changes to each device. This substantially improves performance when writing to devices with high latency such as flash memory and distant network storage.

One can reasonably expect that mobile storage will continue to evolve as new technologies such as MEMS-based storage [16] become available. A file system that embeds static assumptions about device performance may soon be outdated. In contrast, a flexible cache hierarchy offers the possibility of adapting to new devices.

3.2 Power to the people

BlueFS's second design goal is to extend client battery lifetime. Energy is often the critical bottleneck on mobile computers, especially for small devices such as handhelds. On one hand, manufacturers are adding additional capabilities such as ubiquitous network access through 802.11 interfaces, more powerful processors, and larger storage capacities. On the other hand, the amount of energy supplied by batteries is improving slowly [10]. Despite this disconnect, mobile computers have been able to maintain reasonable battery lifetimes by implementing more efficient power management strategies at all system layers.

The energy cost of ubiquitous data access can be high because distributed file systems make heavy use of power-hungry storage and network devices. However, the energy expended by current file systems is often much higher than necessary, simply because energy-efficiency was not an important goal in their design. In contrast, BlueFS contains several mechanisms that substantially reduce energy consumption.

First, BlueFS considers energy cost in addition to performance when deciding from which device it will read data. Second, BlueFS aggregates write operations to amortize the energy cost of spinning up disk drives and powering on network interfaces.

Third, BlueFS integrates with device power management strategies. It uses self-tuning power management interfaces [1, 2] to disclose hints about the devices that it intends to access. The operating system uses these hints to proactively transition devices to save time and energy.

Finally, BlueFS enables the use of more aggressive power management by masking the performance impact of device transitions. For example, many users disable disk power management due to the annoying delays that occur when accessing data after the disk has spun down. BlueFS hides these delays by initially fetching data from the network while it spins up the disk. By limiting the perceived performance impact of power management, BlueFS may increase the adoption of more aggressive power management policies.

3.3 You can take it with you

Our last design goal is transparent support for portable storage such as USB keychains and mobile disks. The capacity of these devices is currently several gigabytes and is growing rapidly. Given a wide-area connection that offers limited bandwidth or high latency, accessing data on portable storage is substantially faster than reading the data from a file server.

Yet, portable storage is not a panacea. Most users currently manage their portable storage with manual copying or file synchronization tools [22]. These methods

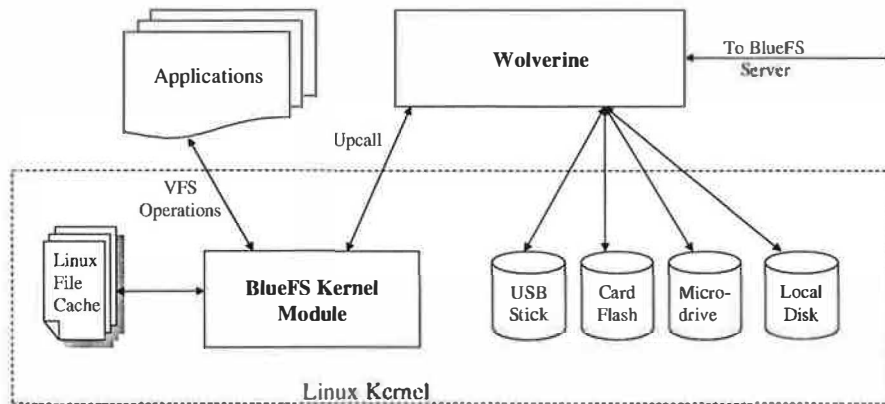


Figure 1. Blue File System architecture

have three potential disadvantages. First, since files on portable storage are not automatically backed up, vital data may be lost if a portable device is misplaced, damaged, or stolen. Second, data on portable storage cannot be accessed from remote computers or shared with other users when the device is not attached to a computer with a network connection. Third, the potential for accessing out-of-date versions or creating inconsistent copies of a file increases greatly when the user manually manages the consistency of file replicas.

BlueFS addresses these problems by having each portable device act as a persistent file cache. BlueFS presents a single system image across all computers and portable storage devices. The primary replica of each block of data resides on the file server. Client and portable storage devices store second-class replicas that are used only to improve performance and reduce energy usage. No data is lost when a portable device is lost, stolen, or damaged since the primary replica always resides on the server.

Blocks cached on portable storage devices are accessible through the BlueFS namespace. When a new device is inserted into a computer, BlueFS adds the device to its search path. On subsequent read operations, BlueFS transparently fetches files from the portable device if that device offers better performance and lower energy usage than other storage options.

When files are modified, BlueFS asynchronously writes modifications to all storage devices attached to the computer, as well as to the file server. Propagation of data to the file server greatly reduces the chance of data loss since files never reside solely on a portable device. Further, files are available from the server even when the portable device is disconnected. Of course, mobile users may wish to access cached data when disconnected from the network. In this case, BlueFS uses techniques pioneered by Coda [9] to allow disconnected operation.

For example, consider a user with a USB keychain who accesses data on computers at home and at work. When the user creates the file `/bfs/myfile` at work, BlueFS transparently replicates it on the keychain. The user brings the keychain home and inserts it into the computer there. When the file is next accessed, BlueFS fetches it from the keychain instead of from a distant server. The only change that the user notices is that performance is substantially better using the keychain. If the user leaves the keychain at work by mistake, the file is retrieved from the file server instead.

Compared to manual copy, BlueFS greatly reduces the potential for data inconsistency by making individual storage devices first-class entities in the file system. The BlueFS file server maintains per-device callbacks for each file. On modification, the server sends an invalidation to the client to which the portable device is currently attached. If the device is disconnected, the invalidation is queued and delivered when the device is next attached to a client. Note that BlueFS cannot eliminate all inconsistencies because it allows disconnected operation and writes modifications asynchronously to improve performance and energy efficiency. Concurrent updates to the same file may create conflicts where two versions of the file exist. As in many other systems that support weaker consistency, such conflicts are automatically detected and flagged for manual or automatic resolution [9].

Manual copy guarantees that a replica of the file, no matter how out of date, will exist on a device until it is deleted. BlueFS provides a stronger guarantee through *affinity*, which lets users specify that a particular device should always store the latest version of a file or subtree. When a file with affinity is invalidated, BlueFS automatically fetches the new version and stores it on the device.

4 Implementation

4.1 Overview

Figure 1 shows the BlueFS architecture. To simplify implementation, we perform the bulk of the client functionality in a user-level daemon called Wolverine. A minimal kernel module, described in the next section, intercepts VFS calls, interfaces with the Linux file cache, and redirects operations to Wolverine. Wolverine, described in Section 4.3, handles reading and writing of data to multiple local, portable, and remote storage devices. The BlueFS server, described in Section 4.4, stores the primary replica of each object (i.e. each file, directory, symlink, etc.)

BlueFS associates a 96-bit identifier with each object: the first 32 bits specify an administrative volume [8], and the remaining 64 bits uniquely identify the object within that volume. Caching in BlueFS is performed at the block-level. Operations that read or modify data operate on whole 4 KB blocks — this size was chosen to match the Linux page cache block size. As described in Section 4.3.6, cache consistency is maintained with version stamps and callbacks [8].

4.2 BlueFS kernel module

Similar to many previous file systems [8, 9], BlueFS simplifies implementation by performing most functionality in a user-level daemon. The BlueFS kernel module intercepts Linux VFS operations and redirects them to the daemon through an upcall mechanism. The kernel module also handles caching and invalidation of data and metadata in the Linux file cache. Operations that modify data such as `write`, `create` and `rename` are redirected synchronously to the daemon. This allows the daemon to support devices with different consistency semantics: for example, the daemon might reflect modifications to the server immediately to minimize the chance of conflicts, but it might delay writing data to disk to minimize expensive disk power mode transitions. The cost of this implementation is that a dirty block is double-buffered until it is written to the last storage device or evicted from the file cache. We plan to eliminate double-buffering in the future by moving more functionality into the kernel.

4.3 The Wolverine user-level daemon

4.3.1 Write queues

Wolverine maintains a write queue in memory for each local, portable, and network storage device that is currently accessible to the mobile computer. Initially, it creates a write queue for the server and one for each permanent storage device on the client. When a portable storage device is attached, Wolverine creates a new queue; when a portable device is detached,

Wolverine deletes its associated queue. Write queues serve two purposes: they improve performance by asynchronously writing data to storage, and they improve energy-efficiency by aggregating writes to minimize expensive power mode transitions.

A write queue is essentially an operation log for a storage device. Whenever Wolverine receives a modification via an upcall, it adds a new record containing that modification to each write queue. Operations are written to storage in FIFO order to guarantee serializability; e.g. since modifications are reflected to the server in order, other clients never view inconsistent file system state.

BlueFS conserves memory by sharing records between write queues. Each record is reference counted and deleted when the last queue dereferences the record. For each queue, Wolverine maintains a per-file hash table of objects that have been modified by queued operations. Before reading data from a device, Wolverine checks its hash table to identify any operations that modify the data and have not yet been reflected to the device.

Each storage device specifies the maximum time that records may remain on its write queue before being written to the device. By default, BlueFS uses a maximum delay of 30 seconds for all write queues (equivalent to the default write-back delay for dirty pages in the Linux file cache). The user may change this value for each device.

Wolverine caps the maximum memory usage of all write queues to a user-specified value. New operations that would cause Wolverine to exceed this limit are blocked until memory is freed by writing queued operations to storage. To avoid blocking, Wolverine starts to flush write queues when memory utilization exceeds 75% of the allowed maximum. Each write queue may be explicitly flushed using a command-line tool.

Wolverine creates bursty device access patterns with a simple rule: whenever the first operation in a queue is written to storage, all other operations in that queue are flushed at the same time. This rule dramatically reduces the energy used by storage devices that have large transition costs between power modes.

For example, disk drives save energy by turning off electronic components and stopping platter rotation. Entering and leaving power-saving modes consumes a large amount of energy that is only recovered if the disk stays powered down for several seconds [5]. By burst-writing modifications, BlueFS creates long periods of inactivity during which the disk saves energy.

Write queues also reduce the energy used to write to remote storage. Wireless 802.11 networks use a power-saving mode (PSM) that disables the client interface during periods of light network utilization. However, the mobile computer expends more energy to transmit data in PSM [1] because PSM decreases throughput and

increases latency. Ideally, the wireless network could achieve the best of both worlds by entering PSM during idle periods and disabling PSM before transmitting data. Unfortunately, since the time and energy cost of mode transitions is large, it is not cost effective to disable PSM before a small transmission and re-enable it after. BlueFS solves this problem by aggregating many small transmissions into a single large one — this makes power-mode toggling effective.

It seems counter-intuitive that writing data to more than one storage device could save energy. However, we find that writing data to multiple devices creates opportunities to save energy in the future. For example, if the only copy of an object exists on a disk drive, then that disk must be spun up when the object is next read. In contrast, if the object is also written to a remote server, then the file system may avoid an expensive disk transition by reading data from the server. Further, since writes can be aggregated to minimize the number of transitions while reads usually cannot, writing to multiple devices in BlueFS tends to *reduce* the total number of transitions that occur. When transitions are the dominant component of energy usage, this saves energy.

4.3.2 Reading data

The kernel module sends Wolverine an upcall whenever data it needs is not in the Linux file cache. Wolverine then tries to read the data from the storage option that offers the best performance and lowest energy cost.

Wolverine maintains a running estimate of the current time to access each storage device. Whenever data is fetched, Wolverine updates a weighted average of recent access times, as follows:

$$\text{new est.} = (\alpha)(\text{this msmt.}) + (1 - \alpha)(\text{old est.})$$

For the server, latency and bandwidth are estimated separately. For local storage, separate estimates are used for the first block accessed in a file (this captures disk seek time) and contiguous block accesses in the same file. Since storage behavior tends to be more stable than network access times, BlueFS sets α to 0.1 for network storage and 0.01 for local storage. We chose these values based on empirical observation of what worked best in practice.

One might reasonably expect that passive monitoring of access times alone would yield good estimates. However, we have found that monitoring device power states is also essential. In fact, the power state is often the dominant factor in determining the latency of the next access. For example, the time to read data from a Hitachi micro-drive increases by approximately 800 ms when the drive is in standby mode. Also, server latency increases by up to 100 ms when the network interface enters its power-saving mode.

BlueFS passively monitors *both* access times and device power modes. It uses the OS power manager described in [2] to receive a callback whenever a network or storage device transitions to a new power mode. For each storage device, it maintains estimators for each possible power mode. When predicting access times, BlueFS uses the estimator that matches the current mode.

In addition to performance, Wolverine also considers energy usage when deciding which device to access. Because on-line power measurement is often infeasible, BlueFS relies on offline device characterization to predict energy costs. These characteristics need be determined only once for each type of I/O device — they are provided when the device is first registered with BlueFS. Often, relevant energy characteristics can be found in device specifications. Alternatively, we have developed an offline benchmarking tool that measures the energy used to read data, write data, and transition between power modes [1]. Given a device characterization, BlueFS predicts the energy used *by the entire system* to access data. This includes not just energy used by device I/O, but also the energy expended while the mobile computer waits for the request to be serviced.

The cost of accessing each device is the weighted sum of the predicted latency and energy usage. By default, BlueFS weights these two goals equally; however, the user may adjust their relative priority. BlueFS first tries to read the data from the least costly device. If the data is not stored on that device, BlueFS then tries the remaining devices in order. Since the server has the primary replica of each file, Wolverine will only fail to read the data on any device when the client is disconnected.

Before reading an object from a device, Wolverine checks the device's write queue hash table to see if operations that modify that item have not yet been reflected to storage. If this is the case, the request can often be serviced directly from the queued operation. For example, a queued store operation may contain the block of data being read. Sometimes Wolverine must replay queued operations that modify the requested object. For example, if the write queue contains operations that modify a directory, Wolverine first fetches the directory from the device, then replays the modifications in memory. Finally, it returns the modified directory to the kernel.

4.3.3 The enode cache

Early in our implementation, we found that BlueFS sometimes wasted time and energy trying to read data from a storage device that did not contain the needed data. We therefore considered how to minimize the number of fruitless reads that occur. We explored maintaining an in-memory index of cached data for each device. However, given the limited memory and rapidly increasing storage capacity of many small mobile devices, we

felt that such an index would often be too large. Instead, we decided to strike a reasonable compromise by caching index information only for recently accessed files.

When BlueFS first accesses an object, it creates an *enode* that captures all the information it currently holds about the validity of that object on the storage devices attached to the mobile computer. For each device, the enode indicates whether the object's state is known or unknown. For an object with known state, the enode tells whether its attributes are valid, and whether none, some, or all of its data blocks are cached. Enodes are hashed by file id and stored in an *enode cache* managed by LRU replacement. The default size of the cache is 1 MB.

Wolverine checks the enode cache before trying to read an object from a device. It skips the device if an enode reveals that the object is not present or invalid. Whenever Wolverine tries to access an object on a new device, it updates the object's enode with its current status (i.e. whether it is present on the device, and if so, whether its data and attributes are valid).

4.3.4 Storage devices

Wolverine supports a modular VFS-style device interface for storage. Each device provides methods to read data and attributes, as well as methods to handle queued modifications. We have implemented two devices that use this interface. The *remote* device communicates with the server through a TCP-based RPC package. The *cache* device performs LRU-style caching of object attributes and data blocks.

The cache device is layered on top of a native file system such as ext2. This simplifies implementation and allows BlueFS and non-BlueFS files to co-exist on a single local or portable storage device. Thus, a portable storage device with a BlueFS cache may also be used on computers that are not BlueFS clients; however, such computers would only access the non-BlueFS files.

Each object and its attributes are stored in a single container file in the native file system that is named by the object's 96-bit BlueFS id. This implementation sacrifices some efficiency, but has let us quickly support a wide variety of storage media. Data blocks are cached in their entirety, but not all blocks in a file need be cached. Similar to the enode structure, each container file has a header that specifies whether the object attributes are valid, and whether its data is invalid, partially valid, or entirely valid. When only a portion of the data blocks are cached, a bit mask determines which blocks are valid.

To start using a new storage device, a client registers the device with the BlueFS server. The server assigns a unique id that is written to a metadata file on the storage device. During registration, the user also specifies the maximum amount of storage that BlueFS may use on that

device. After a device is registered, it may be attached to any BlueFS client.

If the cache on a device becomes full, BlueFS evicts objects using the second-chance clock algorithm, with the additional modification that object metadata is given additional preference to remain in the cache. Wolverine starts the eviction thread whenever a cache becomes 90% full. Since the primary replica of a file always exists on the server, the eviction thread simply deletes container files to create space.

To maintain the LRU approximation, Wolverine ensures that each cache is updated with the latest version of data being read by the user. When data is read from the server or from another device, Wolverine checks the object's enode to determine if the cache already has the latest version of the data. If the cache does not have the data or its status is unknown, Wolverine places a copy of the data on the device's write queue. This ensures that each local and portable device has up to date copies of files in the user's current working set.

4.3.5 Affinity

While LRU replacement is a convenient way to capture a user's current working set on a local or portable device, it is often the case that a user would prefer to always have a certain set of files on a particular device. In BlueFS, device *affinity* provides this functionality.

Affinity specifies that the latest version of an object should always be cached on a particular device. Although affinity is similar in spirit to hoarding [11] in the Coda file system, affinity can support portable devices because it is implemented at device granularity. In contrast, Coda implements hoarding at client granularity, meaning that one cannot hoard files to portable storage or to multiple devices on the same client.

In BlueFS, a command-line tool provides users with the ability to add, display, or remove affinity for files and subtrees. Affinity is implemented as an attribute in the cache container file header. If the affinity bit is set, the object is never evicted. If a file with affinity is invalidated by a modification on another client, Wolverine adds the object to a per-device *refetch list*. Every five minutes, a Wolverine thread scans the refetch lists of all attached devices and requests missing data from the server. If the set of files with affinity to a device is greater than the size of the cache, the user must remove affinity from some files before the thread can refetch all files.

Affinity for subtrees is supported with a *sticky affinity* attribute that is associated with directories. Whenever an object is added as a child of a directory with sticky affinity to a device, the new object is given affinity to that device. If the new object is a directory, it also receives sticky affinity. For example, if a user specifies that

a PowerPoint directory has sticky affinity to a USB stick, new files created in that directory will also have affinity to the USB stick.

4.3.6 Cache consistency

Cache consistency in BlueFS builds upon two pieces of prior work: Coda's use of optimistic replica control [9] and AFS's use of callbacks for cache coherence [8]. BlueFS adds two important modifications. First, BlueFS maintains callbacks on a per-device basis, rather than on a per-client basis. Second, the BlueFS server queues invalidation messages when a device is disconnected. These modifications let BlueFS efficiently support portable media and clients that frequently hibernate to save energy.

Optimistic replica control provides greater availability and improves performance by allowing clients to read or write data without obtaining leases or locks. When two clients concurrently modify a file, the write/write conflict is detected at the server and flagged for manual resolution. Like Coda, BlueFS stores a version number in the metadata of each object. Each operation that modifies an object also increments its version number. Before committing an update operation, the server checks that the new version number of each modified operation is exactly one greater than the previous version number. If this check fails, two clients have made conflicting modifications to the object. The user must resolve such conflicts by selecting a version to keep. In the absence of disconnection, such conflicts occur only when two different clients update the same file within 30 seconds of each other (i.e. the amount of time an update can reside in the server write queue). Prior experience in other distributed file systems [9] has shown that such conflicts are rare.

Callbacks are a mechanism in which the server remembers that a client has cached an object and notifies the client when the object is modified by another client. In contrast to AFS and Coda, BlueFS maintains callbacks on a per-device rather than a per-client granularity. Specifically, a BlueFS device may hold a data and/or a metadata callback for each object. When Wolverine adds an operation to a device write queue that caches data, it also adds a callback request on behalf of that device to the server write queue. It uses the enode cache to reduce the number of callback requests. The object enode records the set of devices for which the server is known to hold a callback. When a callback is already known to exist for a device, no request need be sent to the server. Wolverine also sets callbacks for objects stored in the Linux file cache. From the point of view of the server, a client's file cache is simply another device.

Each callback request contains the version number of the cached object. If this does not match the version

number of the primary replica, the server issues an immediate invalidation. Otherwise, the server maintains the callback as soft state.

Wolverine optimizes invalidation delivery by notifying the server when storage devices are attached and detached. The server does not send invalidations to devices that are currently attached to the client that made the modification since the client will already have placed the modification on each device's write queue. The server sends invalidations to all other devices that have a callback on the object; invalidations for multiple devices attached to the same client are aggregated.

Upon receipt of an invalidation, the client updates the object enode to indicate that the data and/or metadata are no longer valid. Invalidations for the Linux file cache are passed to the kernel module, which removes the data from the file cache. Other invalidations are placed on device write queues. When the queues are flushed, invalidated objects are deleted. Note that the presence of the invalidation record in a queue's hash table prevents stale data from being read before the queue is flushed.

As described, callbacks provide cache consistency only when the server and the client to which a storage device is attached remain continuously connected. During disconnection, invalidation messages could potentially be lost, so the validity of cached objects becomes unknown. In our initial design, BlueFS performed *full cache revalidation* upon reconnection. During revalidation, BlueFS verifies each cached object by asking the server for the version number and identifier of the last client to make an update. If these values match those of the cached replica, the replica is valid and a new callback is set. Otherwise, the replica is deleted.

However, due to the growing size of mobile storage devices, full cache validation is expensive in terms of both time and energy. Unfortunately, mobile storage devices are frequently disconnected from the server. First, portable devices are naturally disconnected when they are detached from one computer and reattached to another. Second, handheld and other mobile computers frequently hibernate or enter other power saving modes in which the network interface is disabled. Thus, the devices specifically targeted by BlueFS frequently disconnect and would often require full cache revalidation given our initial design.

We therefore modified our design to have the server queue invalidation messages for later delivery when a device holding a callback is disconnected. When a portable device is next attached to a BlueFS client, the server forwards all invalidations queued during disconnection to the new client. Similarly, when a client hibernates to save power, the server queues invalidations for that client's file cache and all its attached devices. This lets hand-

held clients nap for short periods without missing invalidations. With these modifications, BlueFS avoids full cache revalidations in most cases. A revalidation is required only when soft state is lost due to client or server crash, or when a portable storage device is not cleanly detached as described in the next section.

From the server's point of view, client disconnection is equivalent to hibernation. When Wolverine detects that the server is unreachable, it writes operations that are flushed from the server write queue to a file on a user-specified local (non-portable) storage device. Upon reconnection, it reads the file and sends these operations to the server before transmitting new updates.

4.3.7 Adding and removing storage devices

Once a portable storage device is registered with the server, it may be dynamically attached and detached from any BlueFS client.

When a portable device is detached, Wolverine:

- sends a detach message to the server. The server starts queuing callbacks at this point.
- stops placing new operations on the device's write queue. This is done atomically with processing of the detach message.
- retrieves and stores the current version of any objects on the device's affinity-driven refetch list.
- flushes any operations remaining on the write queue to the device
- flushes the server write queue.
- writes to the device the state of the enode cache for that device.
- writes a clean shutdown record to that device.

A clean detach guarantees that the latest version of all files with affinity to the device are stored on that device. BlueFS also supports an option in which the detach does not refetch files from the server, but instead records the list of files to be refetched in a file on the device. This option makes detach faster but these files are not available if the device is next attached to a disconnected client.

When a portable device is attached, Wolverine:

- checks for a clean shutdown record. If the record is found, it is deleted. If it is not found, full cache revalidation is performed.
- sends an attach message to the server.
- deletes any objects invalidated by queued callbacks contained in the server's response to the attach message.
- warms the enode cache with the device's previously saved state. New enodes are created if the cache is not full; otherwise, only enodes in the cache are updated with the device state.

- reschedules any refetch requests that were saved during detach.

If a device was not cleanly detached, it must be fully revalidated on reattachment. Currently, revalidation must also be performed when devices are detached from a disconnected client.

4.3.8 Fault tolerance

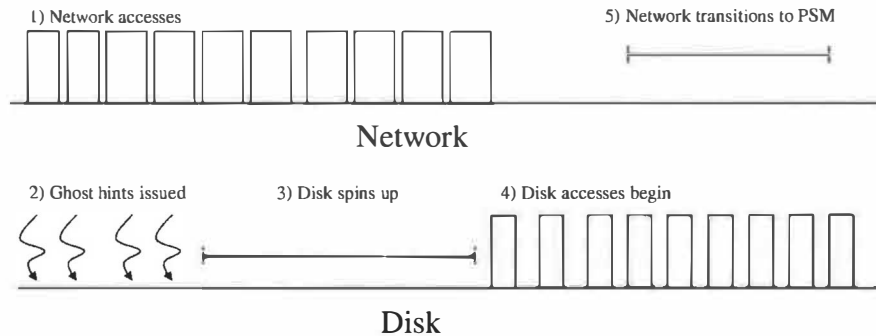
The BlueFS server writes modifications to a write-ahead log before replying to RPCs. Since modifications are safe only after the server replies, BlueFS clients can lose approximately 30 seconds worth of updates if they crash (i.e. the set of updates that resides in the server write queue). In the common case, no data is lost when a local or portable storage device on the client suffers a catastrophic failure (since the primary replica of each object exists on the server). However, modifications made during disconnected operation are lost if the device holding the persistent head of the server write queue fails. Although BlueFS does not currently guard against catastrophic server failure, well-known mechanisms such as server or storage replication could provide this functionality.

4.3.9 Integration with power management

BlueFS works closely with device power management to minimize energy usage. It discloses hints about its I/O activity using self-tuning power management (STPM). Since STPM has been described elsewhere [1], we present only a brief description here.

BlueFS issues a STPM hint every time it performs an RPC. The hint contains the size of the RPC and discloses whether the RPC represents background or foreground activity. The STPM module uses this information to decide when to enable and disable power-saving modes. This decision is based upon a cost/benefit analysis that considers the observed distribution of RPCs, the base power of the mobile computer, and the user's current relative priority for performance and energy conservation. For example, if BlueFS performs three RPCs in close succession, STPM might anticipate that several more RPCs will soon occur and disable power management to improve performance.

If an adaptive system such as BlueFS adopts a purely *reactive* strategy that considers only one access at a time, it will often make incorrect decisions about which device to access. As an example, consider the case where BlueFS is reading many small files stored on the server and on a disk in standby mode. The best strategy is to fetch the files from disk, since the transition cost of spinning up the disk will be amortized across a large number of accesses. However, a system that considers only one access at a time will never spin up the disk since the transition cost always outweighs the benefit that will be



BlueFS masks the latency of disk power management by initially fetching data from the server. Ghosts hints are issued to the disk, which transitions to active mode. After the transition, BlueFS fetches the remaining data from disk. Eventually, the idle network device transitions to PSM to save power.

Figure 2. Masking the latency of disk power management

realized by a single file. Dynamic power management cannot help here: the disk will never spin up because the OS sees no disk accesses.

BlueFS solves this problem using *ghost hints* [2] that disclose to the OS the opportunity cost that is lost when a device is in a power-saving mode. When Wolverine predicts the time and energy cost of accessing each device in its current power mode, it also predicts the cost that would have resulted from that device being in its ideal mode. If the device that has the lowest current cost differs from the device that has the lowest ideal cost, Wolverine issues a ghost hint to the device with lowest ideal cost. The ghost hint discloses the time and energy wasted by that device being in the wrong power mode. For example, when the disk STPM module receives a set of ghost hints whose lost opportunity cost exceeds the cost of the disk transition, it spins up the disk. Thus, ghost hints enable BlueFS to *proactively* switch to the best device during periods of high load.

We have designed BlueFS to mask the performance impact of device power management. Consider the case shown in Figure 2 where BlueFS is fetching a large file with the disk initially in standby mode to conserve power. Using other file systems, fetching the first block incurs a large delay as the disk spins up (800 ms for the Hitachi microdrive [7] and one second for the IBM T20 laptop drive). In contrast, BlueFS hides this delay by fetching blocks over the network. For each block that it fetches from the network, BlueFS issues a ghost hint that discloses the opportunity cost of the disk being in standby. While the disk spins up in response to the ghost hints, BlueFS continues to fetch blocks from the server. After the transition completes, BlueFS fetches remaining blocks from disk. Eventually, the network device enters a power-saving mode since it is now idle. No ghost hints are issued to the network device because the disk offers both the lowest current and ideal cost.

Our belief is that many users avoid disk power management because of lengthy spin up delays. If BlueFS hides these delays, then users might use more aggressive power management policies. However, the potential energy savings are extremely difficult to quantify; if such savings were to occur, they would supplement those we report in Section 5.5.

4.4 BlueFS server

Since the focus of BlueFS is on the mobile client, our file server implementation is fairly standard. We expect a single BlueFS server to handle roughly the same number of clients handled by a current AFS or NFS server. The BlueFS server stores the primary replica of every object. When a client modifies an object, it sends the modification to the server. The server provides atomicity and durability for file system operations through write-ahead logging and serializability by locking the BlueFS identifier namespace. The largest difference between the BlueFS server and those of other file systems is that the BlueFS server is aware of individual storage devices.

5 Evaluation

Our evaluation answers five questions:

- Do the new features of BlueFS hurt performance during traditional file system benchmarks?
- How well does BlueFS mask the access delays caused by disk power management?
- How well does BlueFS support portable storage?
- How much does BlueFS reduce energy usage?
- Can BlueFS adapt to storage devices with heterogeneous characteristics?

5.1 Experimental setup

We used two client platforms: an IBM T20 laptop and an HP iPAQ 3870 handheld. The laptop has a 700 MHz Pentium III processor, 128 MB of DRAM and a 10 GB hard drive. The handheld has a 206 MHz StrongArm processor, 64 MB of DRAM and 32 MB of flash. The laptop runs a Linux 2.4.28-30 kernel and the handheld a Linux 2.4.18-rmk3 kernel. Unless otherwise noted, both clients use a 11 Mb/s Cisco 350 802.11b PCMCIA adapter to communicate with the server. The portable device in our experiments is a 1 GB Hitachi microdrive [7].

The server is a Dell Precision 350 desktop with a 3.06 GHz Pentium 4 processor, 1 GB DRAM, and 120 GB disk, running a Linux 2.4.18-19.8.0 kernel. The client and server communicate using a Cisco 350 802.11b base station. When we insert delays, we route packets through an identical Dell desktop running the NISTnet [4] network emulator.

We measure operation times using the `gettimeofday` system call. Energy usage is measured by attaching the iPAQ to an Agilent 34401A digital multimeter. We remove all batteries from the handheld and sample power drawn through the external power supply approximately 50 times per second. We calculate system power by multiplying each current sample by the mean voltage drawn by the mobile computer — separate voltage samples are not necessary since the variation in voltage drawn through the external power supply is very small. We calculate total energy usage by multiplying the average power drawn during benchmark execution by the time needed to complete the benchmark. The base power of the iPAQ with network in PSM and disk in standby is 1.19 Watts.

We limit maximum write queue memory usage for BlueFS to 50 MB on the laptop and 15 MB on the handheld. Unless otherwise noted, we assign an equal priority to energy conservation and performance.

5.2 Modified Andrew benchmark

We begin our evaluation with the traditional modified Andrew benchmark. While this benchmark does not have a workload typical of how we expect BlueFS to be used, its widespread adoption in the file system community makes it a useful data point. We compare three file systems: NFS version 3, operating in asynchronous mode; Coda version 6.0.3 running in both write connected mode, in which modifications are reflected synchronously to the server, and write disconnected mode, in which modifications are reflected asynchronously; and BlueFS. We measure the time needed by each file system to untar the Apache 2.0.48 source tree, run `configure`, make the executable, and delete all source and object files. In total, the benchmark generates 161 MB of data.

Latency (ms)	NFS	Coda		BlueFS
		Wr. Conn.	Wr. Disc.	
0	879 (3)	702 (3)	601 (15)	552 (11)
30	5779 (56)	1681 (4)	637 (10)	535 (14)

This figure shows the number of seconds needed to untar, configure, make, and delete the Apache 2.0.48 source tree. Each value shows the mean of 5 trials with standard deviation given in parentheses.

Figure 3. Modified Andrew benchmark

Figure 3 shows the result of running this benchmark on the IBM T20 laptop. With no network latency between client and server, BlueFS executes the benchmark 59% faster than NFS and 9% faster than Coda in write disconnected mode. With a 30 ms latency, BlueFS is over 10 times faster than NFS and 19% faster than Coda.

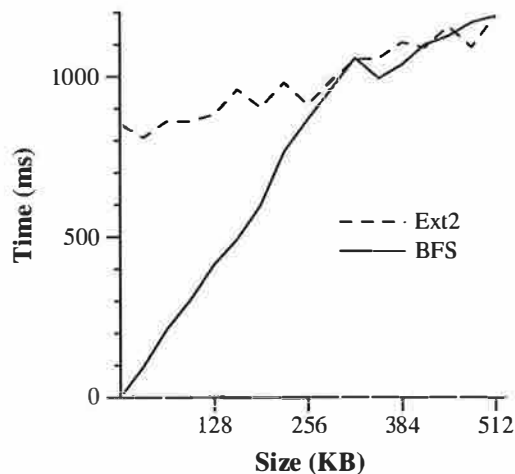
BlueFS and Coda both outperform NFS because they cache data on the laptop hard drive. In contrast, NFS must refetch data from the server during benchmark execution — this is especially costly when latency is high. The difference in Coda's performance between write connected and write disconnected mode shows the benefit of writing modifications asynchronously to the server. Given this performance difference, we operate Coda in write disconnected mode in subsequent experiments.

Originally, we had expected BlueFS to exhibit performance similar to that of Coda in write disconnected mode. One possible reason for the performance advantage seen by BlueFS is its use of TCP rather than Coda's custom transport protocol. Another possible reason is Coda's use of RVM. However, we note that Coda's resilience to client crash in write disconnected mode is similar to that of BlueFS — up to 30 seconds of data can be lost.

5.3 Masking the latency of power management

We next examined how well BlueFS hides the access delays caused by disk power management. We first measured the time used by the Linux ext2 file system to read files from the Hitachi 1 GB microdrive when the drive is in standby mode. As shown by the dashed line in Figure 4, ext2 reads are delayed for approximately 800 ms while the drive spins up to service the first request. Note that distributed file systems such as Coda pay the same performance penalty as ext2 because they use a static cache hierarchy that always fetches data from disk.

We next measured the time to read the same files using BlueFS. BlueFS masks disk spin-up delay by reading the first few blocks of the file from the server. For small files, this leads to a substantial performance improvement; BlueFS reads a 4 KB file in 13 ms, whereas ext2 requires over 800 ms. This gap may be larger for other hard drives — our laptop hard drive spin-up delay is greater than 1 second.



This figure shows the time used by ext2 and BlueFS to read 4–512KB files. Files are stored on a portable Hitachi 1 GB microdrive and on the BlueFS server. At the beginning of each trial, the network is in CAM and the microdrive is in standby mode. No latency is inserted between the BlueFS client and server. Each value is the mean of five trials.

Figure 4. Masking power management latency

The performance benefit of BlueFS decreases with file size. For large files, BlueFS ghost hints spin up the disk. During the transition, BlueFS fetches blocks from the network. Although BlueFS improves performance by fetching the first part of each file from the network, this benefit is offset by the lack of readahead in our current implementation. The break-even file size, at which BlueFS and ext2 have equivalent performance, is 256 KB. However, *first-byte latency* is an important metric for applications such as grep and Mozilla that can begin useful processing before all bytes have been read. Since BlueFS delivers the first block of large files to the application 800 ms faster than ext2, many applications will be able to begin processing data sooner.

5.4 Support for portable storage

We measured how well BlueFS supports portable storage using a benchmark devised by Tolia et al. [20]. The benchmark replays a set of Coda traces collected at Carnegie Mellon University using Mummert’s DFSTrace tool [11]. The traces, whose characteristics are shown in Figure 5, capture file system operations performed over several hours on four different computers. Due to Coda’s open-close semantics, the traces do not capture individual read and write operations. We therefore modified DFSTrace to assume that files are read in their entirety on open. Further, files that are modified are assumed to be written in their entirety on close.

Each file system trace is accompanied by a snapshot of the file system at the start of trace collection. At the beginning of each experiment, the snapshot data exists on the file server and the portable microdrive, but not on

Trace	Number of Ops.	Length (Hours)	Update Ops.	Working Set (MB)
purcell	87739	27.66	6%	252
messiaen	44027	21.27	2%	227
robin	37504	15.46	7%	85
berlioz	17917	7.85	8%	57

This figure shows the file system traces used in our evaluation. Update operations are those that modify data. The working set is the total size of the files accessed during a trace.

Figure 5. File traces used in evaluation

the local disk of the client. We also generate a lookaside index of the snapshot on the microdrive. Before running the benchmark, we flush the Linux file cache on both the client and server.

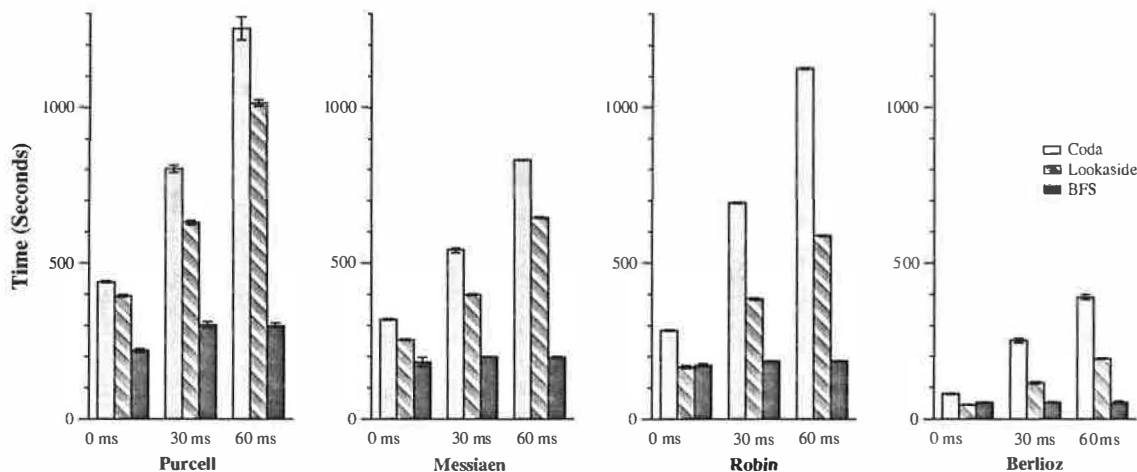
Tolia’s benchmark replays each trace as fast as possible and measures the time to complete all operations. We therefore set BlueFS’s performance/energy knob to maximum performance. We ran the benchmark on the IBM T20 laptop, using a Lucent 2 Mb/s wireless network card to limit network bandwidth. We also use NISTnet to vary network latency from 0 to 60 ms.

Figure 6 compares the time to replay the traces without portable storage using Coda, with portable storage using Coda’s lookaside caching, and with portable storage using BlueFS. At 0 ms latency, BlueFS reduces benchmark execution time 28–44% for the Purcell and Messiaen traces compared to Coda lookaside caching. Although lookaside caching pays little penalty for revalidating objects with the server at this latency, BlueFS achieves a substantial benefit by fetching small files from the server rather than the microdrive. For the Robin and Berlioz traces, BlueFS and lookaside caching perform similarly with no latency. For the four traces at 0 ms latency, BlueFS reduces replay time 34–50% compared to Coda without portable storage.

As latency increases, the performance advantage of BlueFS grows. Lookaside caching requires one server RPC per object in order to retrieve the attributes and SHA-1 hash — BlueFS avoids this round-trip by using device callbacks. At a latency of 30 ms, BlueFS fetches all data from portable storage — thus, it is not affected by further increases in network latency. At this latency, BlueFS is at least twice as fast as Coda with lookaside caching for the four traces.

At 60 ms latency, BlueFS executes the benchmark 5–7 times faster than Coda without lookaside caching and over 3 times faster than Coda with lookaside caching. Over a higher-bandwidth, 11 Mb/s network connection, we saw roughly similar results — BlueFS executed the benchmark approximately 3 times faster than Coda with lookaside caching at a network latency of 60 ms.

Finally, it is important to note that BlueFS updates the microdrive as objects are modified during the trace. At



This figure compares the performance benefit achieved by BlueFS and Coda through the use of portable storage. Each graph shows the time to replay a file system trace with data stored on a local 1 GB microdrive and on a file server with 0, 30, and 60 ms of network latency. Each value is the mean of 3 trials — the error bars show the highest and lowest result.

Figure 6. Benefit of portable storage

the end of trace replay, the microdrive holds a complete version of the modified file tree. Since Coda lookaside caching provides read-only access to portable storage, many cached objects are invalid at the end of trace replay. In order to bring the cache up to date, the user must manually synchronize modified files and generate a new lookaside index. For large caches, index generation is time-consuming (e.g. 3 minutes for the Purcell tree), because it requires a full cache scan.

5.5 Energy-efficiency

We next measured the energy-efficiency of BlueFS by running the Purcell trace on the iPAQ handheld using the microdrive as local storage. In this experiment, it is important to correctly capture the interaction of file system activity and device power management by replicating the interarrival times of file system operations. Thus, between each request, replay pauses for the amount of time recorded in the original trace. Since running the entire trace would take too long using this methodology, we replay only the first 10,000 operations — this corresponds to 42:36 minutes of activity.

In our first experiment, shown in Figure 7, all files accessed by the trace are initially cached on the microdrive. The left graph compares the interactive delay added to trace replay by BlueFS and Coda — this shows the time that the user waits for file operations to complete. The right graph shows the amount of additional energy used by the iPAQ to execute file system operations (i.e. it excludes energy expended during think time). Coda uses default power management (PSM for the network and the microdrive's ABLE power manager). Note that power management is essential to achieve reasonable battery

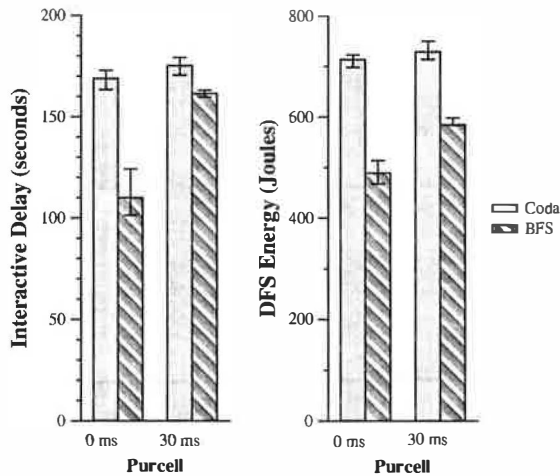
lifetime; Coda runs faster without power management, but the iPAQ expends over 3 times more power.

With 0 ms latency, BlueFS reduces interactive delay by 35% and file system energy usage by 31%. Since all files are on disk, Coda only uses the network to asynchronously reintegrate modifications to the server. BlueFS, however, achieves substantial performance and energy benefit by fetching small objects from the server and by using the network when the disk is in standby.

With 30 ms network latency, the microdrive always offers better performance than the server *when it is active*. In the absence of power management, Coda's static hierarchy would always be correct. However, because BlueFS hides performance delays caused by the drive entering standby mode, it reduces interactive delay by 8% compared to Coda. BlueFS also uses 20% less energy. In addition to the energy reduction that comes from its performance gains, BlueFS saves considerable energy by aggregating disk and network accesses.

In Figure 8, we show results when only half of the files accessed by the trace are initially cached on the microdrive. The set of cached files was randomly chosen and is the same in all experiments.

BlueFS excels in this scenario. Compared to Coda, BlueFS reduces interactive delay by 76% and file system energy usage by 55% with no network latency. With 30 ms latency, BlueFS reduces interactive delay by 59% and file system energy usage by 34%. When some files are not cached, file accesses often go to different devices. While one device is being accessed, the others are idle. During long idle periods, these other devices enter power saving modes, creating considerable variability in access times. Thus, this scenario precisely matches the type of dynamic environment that BlueFS is designed to handle.



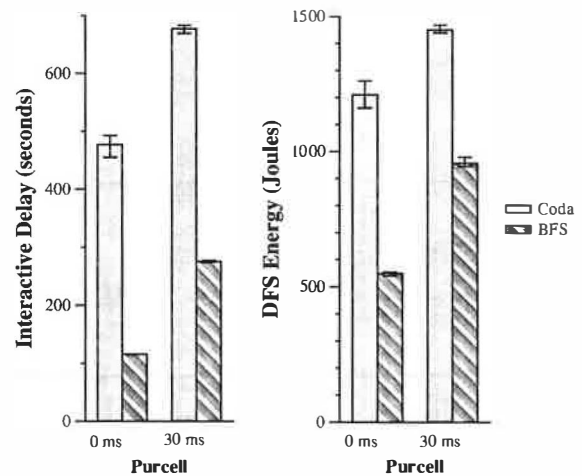
The left graph compares the interactive delay caused by Coda and BlueFS during the execution of the first 10,000 operations in the Purcell trace on an iPAQ handheld. The right graph shows the energy used by each file system. All data is initially cached on a local 1 GB microdrive. Each value is the mean of three trials — the error bars show the highest and lowest result.

Figure 7. Benefit of BlueFS with warm cache

Of course, file system energy usage is only a portion of the total energy expended by the mobile computer. To calculate the effect of using BlueFS on battery lifetime, one must first determine the power used by non-file-system activities. In Figure 8, for example, if one assumes that the iPAQ constantly draws its base power (1.19 Watts) during user think time, use of BlueFS extends battery lifetime by 18% with a 0 ms delay and by 12% with a 30 ms delay. However, the extension in battery lifetime would be much greater if the iPAQ were to hibernate during trace periods with no activity (because average power usage due to non-file-system activity would decrease). Conversely, application activity may increase average power usage and reduce the energy benefit of BlueFS.

5.6 Exploiting heterogeneous storage

In our final experiment, we explored how well BlueFS utilizes multiple storage devices with heterogeneous characteristics. The iPAQ handheld contains a small amount (32 MB) of NOR flash that the familiar Linux distribution uses as the root partition. Reading a 4 KB block from flash is inexpensive: it takes less than 1 ms and uses negligible energy. However, flash writes are very costly — our measurements show that the iPAQ flash has substantially lower write throughput and higher latency than the microdrive. One reason for the low throughput is that modifying a block of flash requires a time-consuming erase operation before the block can be overwritten. Another reason is that the iPAQ uses the jffs2 file system, which performs writes synchronously. Finally, jffs2 is log-structured, and garbage collection



The left graph compares the interactive delay caused by Coda and BlueFS during the execution of the first 10,000 operations in the Purcell trace on an iPAQ handheld. The right graph shows the energy used by each file system. Half of the data is initially cached on a local 1 GB microdrive. Each value is the mean of three trials — the error bars show the highest and lowest result.

Figure 8. Benefit of BlueFS with 50% warm cache

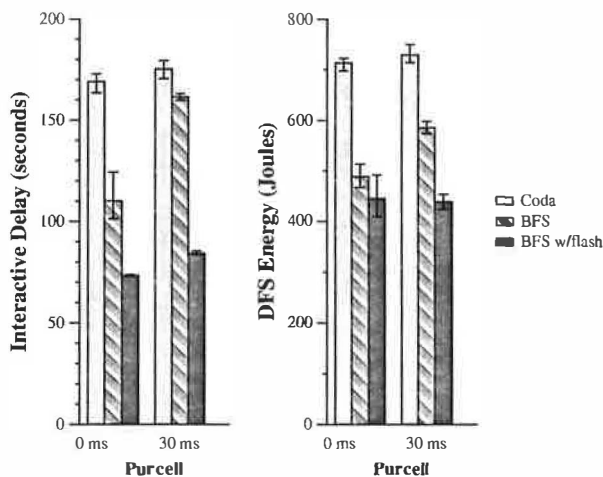
uses substantial time and energy [25].

Despite these obstacles, we decided to see if BlueFS would benefit by using the iPAQ's flash. We could allocate only 16 MB of flash for BlueFS — much less than the working set of the Purcell trace. When we untar the initial tree for the trace on the iPAQ, BlueFS caches all data on the microdrive but only a small portion on flash. At the end of the untar, roughly 10,000 attributes and the last several hundred files accessed are on flash.

Our initial results were disappointing. After investigation, we found the culprit: flash read operations block for up to several hundred milliseconds while one or more flash blocks are erased to make room for new data. Based on this observation, we modified the BlueFS latency predictor to correctly anticipate reduced performance during flash queue flushes. We also increased the maximum delay of the flash write queue to 300 seconds to minimize the number of flushes that occur. We found it interesting that write aggregation proves useful even for a device with no power mode transitions.

With these modifications, BlueFS makes excellent use of the iPAQ flash, as shown in Figure 9. With no network latency, BlueFS with flash reduces interactive delay by 33% compared to our previous results, which used only the microdrive. Use of flash also reduces BlueFS energy usage by 9% on average, despite writing data to an additional device.

With 30 ms latency, flash reduces interactive delay by 48% and BlueFS energy usage by 25%. Observation of individual file operations shows that BlueFS makes the best use of each device: data is read from flash when



The left graph compares the interactive delay caused by Coda and BlueFS during the execution of the first 10,000 operations of the Purcell trace on an iPAQ handheld. The right graph shows the energy used by each file system. All data is initially cached on the microdrive. In addition, 16 MB of data is cached on local flash. Each value is the mean of three trials — the error bars show the highest and lowest result.

Figure 9. Benefit of heterogeneous storage

available, large files are read from the microdrive to utilize its superior bandwidth, and the network is used to read small files and the first block of large files when the disk is spun down to save power.

6 Conclusion

Compared to previous distributed file systems, BlueFS provides three substantial benefits: it reduces client energy usage, seamlessly integrates portable storage, and adapts to the variable access delays inherent in pervasive computing environments. Rather than retrofit these features into an existing file system, we have taken a clean-sheet design approach. The benefits of designing a new system are most apparent in the portable storage experiments: BlueFS substantially outperforms an implementation that modifies an existing file system.

In the future, we plan to broaden the set of caching policies supported on individual devices. For example, on a device that is frequently shared with friends, a user may want to store only files that have affinity to that device. Alternatively, a user might wish to encrypt data on portable devices that could be easily stolen. We believe that BlueFS provides an excellent platform on which to explore this topic, as well as other mobile storage issues.

Acknowledgments

We would like to thank Landon Cox, Niraj Tolia, the anonymous reviewers, and our shepherd, Greg Ganger, for their suggestions that improved the quality of this paper. We would also like to thank Manish Anand for his

help with STPM. This research was supported by the National Science Foundation under awards CCR-0306251 and CNS-0346686, and also by an equipment grant from Intel Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, Intel, the University of Michigan, or the U.S. government.

References

- [1] M. Anand, E. B. Nightingale, and J. Flinn. Self-tuning wireless network power management. In *Proceedings of the 9th Annual Conference on Mobile Computing and Networking*, pages 176–189, San Diego, CA, September 2003.
- [2] M. Anand, E. B. Nightingale, and J. Flinn. Ghosts in the machine: Interfaces for better power management. In *Proceedings of the 2nd Annual Conference on Mobile Computing Systems, Applications and Services*, pages 23–35, Boston, MA, June 2004.
- [3] T. E. Anderson, M. D. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. In *Proceedings of the 15th ACM Symp. on Op. Syst. Principles*, Copper Mountain, CO, Dec. 1995.
- [4] M. Carson. *Adaptation and Protocol Testing thorough Network Emulation*. NIST, <http://snad.ncsl.nist.gov/itg/nistnet/slides/index.htm>.
- [5] F. Douglass, P. Krishnan, and B. Marsh. Thwarting the power-hungry disk. In *Proceedings of 1994 Winter USENIX Conference*, pages 292–306, San Francisco, CA, January 1994.
- [6] T. Heath, E. Pinheiro, and R. Bianchini. Application-supported device management for energy and performance. In *Proceedings of the 2002 Workshop on Power-Aware Computer Systems*, pages 114–123, February 2002.
- [7] Hitachi Global Storage Technologies. *Hitachi Microdrive Hard Disk Drive Specifications*, January 2003.
- [8] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1), February 1988.
- [9] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1), February 1992.

- [10] T. L. Martin. *Balancing Batteries, Power, and Performance: System Issues in CPU Speed-Setting for Mobile Computing*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, 1999.
- [11] L. Mummert, M. Ebling, and M. Satyanarayanan. Exploiting weak connectivity in mobile file access. In *Proceedings of the 15th ACM Symp. on Op. Syst. Principles*, Copper Mountain, CO, Dec. 1995.
- [12] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 174–187, Banff, Canada, October 2001.
- [13] Network Working Group. *NFS: Network File System protocol specification*, March 1989. RFC 1094.
- [14] J. M. Paluska, D. Saff, T. Yeh, and K. Chen. Footloose: A case for physical eventual consistency and selective conflict resolution. In *Proceedings of the 5th IEEE Workshop on Mobile Computing Systems and Applications*, Monterey, CA, Oct. 2003.
- [15] A. E. Papathanasiou and M. L. Scott. Energy efficiency through burstiness. In *Proceedings of the 5th IEEE Workshop on Mobile Computing Systems and Applications*, pages 444–53, Monterey, CA, October 2003.
- [16] S. Schlosser, J. Griffin, D. Nagle, and G. Ganger. Designing computer systems with MEMS-based storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 1–12, Cambridge, MA, November 2000.
- [17] S. Sobti, N. Garg, C. Zhang, X. Yu, A. Krishnamurthy, and R. Wang. PersonalRAID: Mobile storage for distributed and disconnected computers. In *Proceedings of the 1st Conference on File and Storage Technologies*, Monterey, California, pages 159–174, Jan. 2002.
- [18] S. Sobti, N. Garg, F. Zheng, J. Lai, Y. Shao, C. Zhang, E. Ziskind, A. Krishnamurthy, and R. Y. Wang. Segank: A distributed mobile storage system. In *Proceedings of the 3rd Annual USENIX Conference on File and Storage Technologies*, San Francisco, CA, March/April 2004.
- [19] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 172–182, Copper Mountain, CO, 1995.
- [20] N. Tolia, J. Harkes, M. Kozuch, and M. Satyanarayanan. Integrating portable and distributed storage. In *Proceedings of the 3rd Annual USENIX Conference on File and Storage Technologies*, San Francisco, CA, March/April 2004.
- [21] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, T. Bressoud, and A. Perrig. Opportunistic use of content addressable storage for distributed file systems. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 127–140, May 2003.
- [22] A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Department of Computer Science, The Australian National University, Canberra, Australia, 1996.
- [23] A.-I. A. Wang, P. Reiher, G. J. Popek, and G. H. Kuenning. Conquest: Better performance through a disk/persistent-RAM hybrid file system. In *Proceedings of the 2002 USENIX Annual Technical Conference*, Monterey, CA, June 2002.
- [24] A. Weissel, B. Beutel, and F. Bellosa. Cooperative I/O: A novel I/O semantics for energy-aware applications. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 117–129, Boston, MA, December 2002.
- [25] D. Woodhouse. Jffs: The jouralling flash file system. In *Ottawa Linux Symposium*. RedHat Inc., 2001.
- [26] J. Zedlewski, S. Sobti, N. Garg, F. Zheng, A. Krishnamurthy, and R. Wang. Modeling hard-disk power consumption. In *Proceedings of the 2nd USENIX Conference on File and Storage Technology*, pages 217–230, San Francisco, CA, March/April 2003.

Life or Death at Block-Level

Muthian Sivathanu, Lakshmi N. Bairavasundaram,
Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau

*Computer Sciences Department
University of Wisconsin, Madison
{muthian, laksh, dusseau, remzi}@cs.wisc.edu*

Abstract

A fundamental piece of information required in intelligent storage systems is the *liveness* of data. We formalize the notion of liveness within storage, and present two classes of techniques for making storage systems liveness-aware. In the *explicit notification* approach, we present robust techniques by which a file system can impart liveness information to storage through a “free block” command. In the *implicit detection* approach, we show that such information can be inferred by the storage system efficiently underneath a range of file systems, without changes to the storage interface. We demonstrate our techniques through a prototype implementation of a *secure deleting disk*. We find that while the explicit interface approach is desirable due to its simplicity, the implicit approach is easy to deploy and enables quick demonstration of new functionality, thus facilitating rapid migration to an explicit interface.

1 Introduction

“Life is pleasant. Death is peaceful. It’s the transition that’s troublesome.” *Isaac Asimov*

Smarter storage systems need to understand whether blocks are live or dead. Previous work has demonstrated the utility of such knowledge: dead blocks can be used to store rotationally optimal replicas of data [33] or to provide zero-cost writes [31], and failure recovery time can be reduced by restoring only live blocks [23].

Unfortunately, liveness information is not available within modern storage systems, due to the narrow block-based interface between file systems and storage [5, 9]. Storage systems simply observe block-level reads and writes and thus are not aware of logical operations (such as deletes) issued by the file system. This limitation precludes many storage level optimizations [12, 18, 23] and makes others less effective [31, 32, 33].

In this paper, we address this limitation by presenting techniques by which storage systems can be imparted with liveness information. We perform a qualitative and quantitative comparison of two approaches. With *explicit notification*, we augment the interface to storage with a new “free block” command; file systems must be modified to properly use it. With *implicit detection*, we develop techniques to enable the storage system to infer liveness information without any change to the interface.

To better evaluate these approaches, we first formalize the notion of liveness within storage. Specifically, we identify three useful classes of liveness (content, block, and generation liveness), and present techniques for explicit and implicit tracking of each type. Because techniques for imparting liveness information are dependent on the characteristics of the file system, we study a range of file systems, including ext2, ext3 and VFAT; in doing so, we identify key file system properties that impact the feasibility and complexity of such techniques.

To gain more direct experience with liveness-tracking methods, we design, implement, and evaluate a prototype *secure deleting disk* that shreds blocks that have been logically deleted by the file system, making deleted data irrecoverable [12]. We implement secure delete due to its extreme requirements on the type and accuracy of liveness information.

On the surface, both explicit and implicit approaches have their obvious benefits and drawbacks. Explicit notification promises simplicity of implementation but requires broad industry consensus, while implicit detection suggests ease of deployment but at the cost of complexity. Our analysis, however, reveals more complex trade-offs.

We find qualitatively that the explicit approach is less complicated to design and implement. However, while it may appear straightforward to modify file systems to issue “free block” commands, accurate notification in the presence of crashes entails careful integration with file system consistency management schemes, thus noticeably increasing complexity.

We also find that implicit liveness detection is feasible underneath a range of modern file systems; however, some file system behaviors prohibit certain classes of liveness inference. Therefore, we identify the properties that must hold in order to enable or simplify implicit liveness inference. We also propose and implement minor modifications to file systems to conform to those properties.

Finally, we show that implicit liveness detection can be accurate underneath modern asynchronous file systems; our secure delete prototype utilizes implicit liveness to shred blocks that are inferred to be dead. By proving correct operation of implicit secure delete, we demonstrate that implicit liveness can be used in storage applications

with extreme correctness requirements. In evaluating the performance of implicit liveness tracking, we find that it is comparable to the explicit approach.

We conclude that storage systems can more easily implement the explicit approach, if the interface is embellished to support it. However, we see the implicit approach as a complementary instead of competitive technology; because industry consensus on interface change is at best slow-moving, implicit techniques (even if complex) can be of use. Specifically, by deploying a particular technology without explicit interface change, implicit techniques can readily demonstrate possible benefits and thus move industry rapidly towards an explicit change.

The paper is organized as follows. We first present an extended motivation (§2), followed by a taxonomy of liveness (§3), and a list of file system properties that impact techniques for imparting liveness information (§4). We proceed by discussing explicit notification (§5) and implicit detection (§6), and then describe secure deletion (§7). We then describe our initial experience with implicit detection under NTFS, a closed-source file system (§8). Finally, we present a discussion on the relative merits of the implicit and explicit approaches (§9), and finish by discussing related work (§10) and concluding (§11). Appendix A includes a proof of correctness for implicit secure delete.

2 Extended Motivation

In this section, we first present examples of functionality enabled by liveness information, and then motivate two alternative approaches for gathering such information.

2.1 Utility of liveness

Liveness information enables a variety of functionality and performance enhancements within the storage system. Most of these enhancements cannot be implemented at higher layers because they require low-level control available only within the storage system.

Eager writing: Workloads that are write-intensive can run faster if the storage system is capable of *eager writing*, *i.e.*, writing to “some free block closest to the disk arm” instead of the traditional in-place write [8, 31]. However, in order to select the closest block, the storage system needs information on which blocks are live. Existing proposals function well as long as there exist blocks that were never written to; once the file system writes to a block, the storage system cannot identify subsequent death of the block as a result of a delete. A disk empowered with liveness information can be more effective at eager writing.

Adaptive RAID: Information on block liveness within the storage system can also facilitate dynamic, adaptive RAID schemes such as those in the HP AutoRAID system [32]; AutoRAID utilizes free space to store data in RAID-1 layout, and migrates data to RAID-5 when it runs

short of free space. Knowledge of block death can make such schemes more effective.

Optimized layout: Techniques to optimize on-disk layout transparently within the storage system have been well explored. Adaptive reorganization of blocks within the disk [21] and replication of blocks in rotationally optimal locations [33] are two examples. Knowing which blocks are free can greatly facilitate such techniques; live blocks can be collocated together to minimize seeks, or the “free” space corresponding to dead blocks can be used to hold rotational replicas.

Smarter NVRAM caching: Buffering writes in NVRAM is a common optimization in storage systems. For synchronous write workloads that do not benefit much from in-memory delayed writes within the file system, NVRAM buffering improves performance by absorbing multiple overwrites to a block. However, in delete-intensive workloads, unnecessary disk writes can still occur; in the absence of liveness information, deleted blocks occupy space in NVRAM and need to be written to disk when the NVRAM fills up. From real file system traces [20], we found that up to 25% of writes are deleted *after* the typical delayed write interval of 30 seconds, and thus will be unnecessarily written to disk. Knowledge about block death within storage removes this overhead.

Intelligent prefetching: Modern disks perform aggressive prefetching; when a block is read, the entire track in which the block resides is often prefetched [22], and cached in the internal disk cache. In an aged (and thus, fragmented) file system, only a subset of blocks within a track may be live, and thus, caching the whole track may result in suboptimal cache space utilization. Although reading in the whole track is still efficient for disk I/O, knowledge about liveness can enable the disk to selectively cache only those blocks that are live.

Faster recovery: Liveness information enables faster recovery in storage arrays. A storage system can reduce reconstruction time during disk failure by only reconstructing blocks that are live within the file system [23].

Self-securing storage: Liveness information in storage can help build intelligent security functionality in storage systems. For example, a storage level intrusion detection system (IDS) provides another perimeter of security by monitoring traffic, looking for suspicious access patterns such as deletes or truncates of log files [18]; detecting these patterns requires liveness information.

Secure delete: The ability to delete data in a manner that makes recovery impossible is an important component of data security [3, 12, 14]. Government regulations require strong guarantees on sensitive data being “forgotten”, and such requirements are expected to become more widespread in both government and industry in the near future [1]. Secure deletion requires low-level control on block placement that is available only within the storage

system; implementing storage level secure delete requires liveness information within the storage system. We explore secure deletion further in Section 7.

2.2 Acquiring liveness information

Despite the clear benefits of liveness information in storage systems, such information is not currently available. A natural question that arises is how to convey liveness information to storage systems. We discuss two approaches: *explicit notification* and *implicit detection*.

2.2.1 Explicit notification

Explicit notification involves augmenting the existing storage interface with new “allocate block” and “free block” commands, and then modifying file systems to use these commands to explicitly convey liveness information to the storage system. The main benefit of the explicit approach is its potential simplicity; once the new interface is deployed, conveying liveness information is seemingly straightforward.

However, while appearing to be a natural way to achieve our goal, there are a few problems with this approach. First, changing the interface to storage raises legacy issues and requires broad industry consensus. Second, a demand for such a new interface often requires agreement on the clear benefits of the interface, which is difficult to achieve without deployment of the interface - a chicken-and-egg problem.

2.2.2 Implicit detection

Implicit detection is intended to solve the bootstrapping problem in explicit interface evolution. In this approach, the storage system monitors block-level reads and writes issued by the file system from underneath an unmodified interface and infers liveness information implicitly, ideally with no change to the file system above. The implicit approach thus enables demonstration of benefits due to a proposed interface change, thereby making it an evolutionary step towards an eventual interface modification.

Previous work on *semantically-smart* storage systems [2, 23, 24] has explored implicit detection of various forms of file system information from within the storage system, for various storage-level enhancements. The degree of accuracy required from the implicit detection techniques in each case depends on the nature of the application using that information. In X-RAY [2], the storage system utilizes implicit information on file accesses to implement an exclusive storage array cache; inaccurate information in X-RAY simply reduces the potential performance gain. In D-GRAID [23], the storage system utilizes implicit information on the file a block belongs to, in order to place blocks in a fault-isolated fashion, thus improving the availability of the storage system under multiple disk failures; inaccurate information in D-GRAID leads to poor fault isolation, but does not impact correctness because the array still exhibits strictly better

Liveness type	Description	Currently possible?	Example utility
Content	Data within block	Yes	Versioning
Block	Whether a block holds valid data currently	No	Eager write, fast recovery
Generation	Block's lifetime in the context of a file	No	Secure delete, storage IDS

Table 1: Forms of liveness.

availability than traditional RAID. In this paper, we investigate the limits of implicit detection, by considering applications that utilize implicit liveness information in a way that directly impacts correctness.

The primary concern with *implicit interface evolution* is that it ties the interacting layers together. For example, if the file system changes, the storage system will likely need to change as well. However, this issue may not be as problematic as it seems. On-disk formats evolve slowly, for reasons of backwards compatibility. For example, the Linux ext2 file system, introduced in roughly 1994, has had the same layout for its lifetime. Further, the ext3 journaling file system [29] is backwards compatible with the on-disk layout of ext2 and the new extensions to the FreeBSD file system [6] are backwards compatible as well. We also have evidence that commercial storage vendors are already willing to maintain and support software specific to a file system; for example, the EMC Symmetrix storage system [7] comes with software that can understand most common file systems. These trends point to the commercial viability of an implicit detection approach.

3 Liveness in Storage: A Taxonomy

Having discussed the utility of liveness information within a storage system, we now present a taxonomy of the forms of liveness information that are relevant to storage. Such liveness information can be classified along three dimensions: *granularity*, *accuracy*, and *timeliness*.

3.1 Granularity of liveness

Depending on the specific storage-level enhancement that utilizes liveness information, the logical unit of liveness to be tracked can vary. We identify three granularities at which liveness information is meaningful and useful: content, block and generation. A summary is presented in Table 1.

3.1.1 Content liveness

Content liveness is the simplest form of liveness. The unit of liveness is the actual data in the context of a given block; thus, “death” at this granularity occurs on every overwrite of a block. When a block is overwritten with new data, the storage system can infer that the old contents are dead. An approximate form of content liveness is readily available in existing storage systems, and has been explored in previous work; for example, Wang *et al.*’s

virtual log disk frees the past location of a block when the block is overwritten with new contents [31]. Tracking liveness at this granularity is also useful in on-disk versioning, as seen in self-securing storage systems [28]. However, to be completely accurate, the storage system also needs to know when a block is freed within the file system, since the contents stored in that block are dead even without it being overwritten.

3.1.2 Block liveness

Block liveness tracks whether a given disk block currently contains valid data, *i.e.*, data that is accessible through the file system. The unit of interest in this case is the “container” instead of the “contents”. Block liveness is the granularity required for many applications such as intelligent caching, prefetching, and eager writing. For example, in deciding whether to propagate a block from NVRAM to disk, the storage system just needs to know whether the block is live at this granularity. This form of liveness information cannot be tracked in traditional storage systems because the storage system is unaware of which blocks the file system thinks are live. However, a weak form of this liveness can be tracked; a block that was never written to can be inferred to be dead.

3.1.3 Generation liveness

The generation of a disk block is the lifetime of the block in the context of a certain file. Thus, by death of a generation, we mean that a block that was written to disk (at least once) in the context of a certain file becomes either free or is reallocated to a different file. Tracking generation liveness ensures that the disk can detect every logical file system delete of a block whose contents had reached disk in the context of the deleted file. An example of a storage level functionality that requires generation liveness is secure delete, since it needs to track not just whether a block is live, but also whether it contained data that belonged to a file generation that is no longer alive. Another application that requires generation liveness information is storage-based intrusion detection. Generation liveness cannot be tracked in existing storage systems.

3.2 Accuracy of liveness information

The second dimension of liveness is accuracy, by which we refer to the degree of trust the disk can place in the liveness information available to it. Inaccuracy in liveness information can lead the disk into either overestimating or underestimating the set of live entities (blocks or generations). The degree of accuracy required varies with the specific storage application. For example, in delete-squashing NVRAM, it is acceptable for the storage system to slightly overestimate the set of live blocks, since it is only a performance issue and not a correctness issue; on the other hand, underestimating the set of live blocks is catastrophic since the disk would lose valid data. Similarly, in generation liveness detection for secure delete,

it is acceptable to miss certain intermediate generation deaths of a block as long as the latest generation death of the block is known.

3.3 Timeliness of information

The third and final axis of liveness is timeliness, which defines the time between a death occurring within the file system and the disk learning of the death. In the explicit notification approach, if the file system delays “free” notifications (similar to delayed writes), there will be a time lag before the disk learns of a block or generation death. Similarly, in the implicit approach, the periodicity with which the file system writes metadata blocks imposes a bound on the timeliness of the liveness information inferred. In many applications, such as eager writing and delete-aware caching, this delayed knowledge of liveness is acceptable, as long as the information has not changed in the meantime. However, in certain applications such as secure delete, timely detection may provide stronger guarantees.

4 File System Properties

Both explicit and implicit methods for imparting liveness information to storage are dependent on the characteristics of the file system using the storage system. We therefore study the range of techniques required for such liveness notification (or detection) by experimenting underneath three different file systems: ext2, ext3, and VFAT. We have also experimented with NTFS, but only on a limited scale due to lack of source code access; our NTFS experience is described in Section 8. Given that ext2 has two modes of operation (synchronous and asynchronous modes) and ext3 has three modes (writeback, ordered, and data journaling modes), all with different update behaviors, we believe these form a rich set of file systems.

We first begin with a brief background on the various file systems and then outline some high level behavioral properties of a file system that are relevant in the context of liveness information. In the next two sections, we discuss how these properties influence different techniques for storage-level liveness tracking.

4.1 File system background

In this subsection, we provide some background information on the various file systems we study. We discuss both key on-disk data structures and the update behavior.

4.1.1 Common properties

We begin with some properties common to all the file systems we consider, from the viewpoint of liveness tracking. At a basic level, all file systems track at least two kinds of on-disk metadata: a structure that tracks allocation of blocks (*e.g.*, bitmap, freelist), and index structures (*e.g.*, inodes) that map each logical file to groups of blocks.

A common aspect of the update behavior of all modern file systems is *asynchrony*. When a data or metadata

block is updated, the contents of the block are not immediately flushed to disk, but instead, buffered in memory for a certain interval (*i.e.*, the *delayed write interval*). Blocks that have been “dirty” longer than the delayed write interval are periodically flushed to disk. The order in which such delayed writes are committed can be potentially arbitrary, although certain file systems enforce ordering constraints [10].

4.1.2 Linux ext2

The ext2 file system is an intellectual descendant of the Berkeley Fast File System (FFS) [16]. The disk is split into a set of *block groups*, akin to cylinder groups in FFS, each of which contains inode and data blocks. The allocation status (live or dead) of data blocks is tracked through *bitmap blocks*. Most information about a file, including size and block pointers, is found in the file’s inode. To accommodate large files, a few pointers in the inode point to *indirect blocks*, which in turn contain block pointers.

While committing delayed writes, ext2 enforces no ordering whatsoever; crash recovery therefore requires running a tool like *fsck* to restore metadata integrity (data inconsistency may still persist). Ext2 also has a synchronous mode of operation where metadata updates are synchronously flushed to disk, similar to early FFS [16].

4.1.3 Linux ext3

The ext3 file system is a journaling file system that evolved from ext2, and uses the same basic on-disk structures. Ext3 ensures metadata consistency by write-ahead logging of metadata updates, thus avoiding the need to perform an *fsck*-like scan after a crash. Ext3 employs a coarse-grained model of transactions; all operations performed during a certain *epoch* are grouped into a single transaction. When ext3 decides to commit the transaction, it takes an in-memory copy-on-write snapshot of dirty metadata blocks that belonged to that transaction; subsequent updates to any of those metadata blocks result in a new in-memory copy.

Ext3 supports three modes of operation. In *ordered data* mode, ext3 ensures that before a transaction commits, all data blocks dirtied in that transaction are written to disk. In *data journaling* mode, ext3 journals data blocks together with metadata. Both these modes ensure data integrity after a crash. The third mode, *data write-back*, does not order data writes; data integrity is not guaranteed in this mode.

4.1.4 VFAT

The VFAT file system descends from the world of PC operating systems. In this paper, we consider the Linux implementation of VFAT. VFAT operations are centered around the *file allocation table (FAT)*, which contains an entry for each allocatable block in the file system. These entries are used to locate the blocks of a file, in a linked-list fashion. For example, if a file’s first block is at address

Property	Ext2	Ext2+sync	VFAT	Ext3-wb	Ext3-ord	Ext3-data
Reuse ordering		×		×	×	×
Block exclusivity	×	×	×			
Generation marking	×	×		×	×	×
Delete suppression	×	×	×	×	×	×
Consistent metadata				×	×	×
Data-metadata coupling						×

Table 2: File system properties. The table summarizes the various properties exhibited by each of the file systems we study.

b, one can look in entry *b* of the FAT to find the next block of the file, and so forth. An entry can also hold an end-of-file marker or a setting that indicates the block is free. Unlike UNIX file systems, where most information about a file is found in its inode, a VFAT file system spreads this information across the FAT itself and the directory entries; the FAT is used to track which blocks belong to the file, whereas the directory entry contains information like size, type information and a pointer to the start block of the file. Similar to ext2, VFAT does not preserve any ordering in its delayed updates.

4.2 Properties

The update behavior of the file system has a direct influence on the techniques through which liveness information can be imparted to the storage system. Based on our experience with the aforementioned file systems, we identify high-level file system properties that are relevant to liveness tracking. Table 2 summarizes these properties.

Reuse ordering: If the file system guarantees that it will not reuse disk blocks until the freed status of the block (*e.g.*, bitmaps or other metadata that pointed to the block) reaches disk, the file system exhibits *reuse ordering*. This property is necessary (but not sufficient) to ensure data integrity; in the absence of this property, a file could end up with partial contents from some other deleted file after a crash, even in a journaling file system. While VFAT and the asynchronous mode of ext2 do not have reuse ordering, all three modes of ext3, and ext2 in synchronous mode, exhibit reuse ordering.

Block exclusivity: *Block exclusivity* requires that for every disk block, there is at most one dirty copy of the block in the file system cache. It also requires that the file system employ adequate locking to prevent any update to the in-memory copy while the dirty copy is being written to disk. This property holds for certain file systems such as ext2 and VFAT. However, ext3 does not conform to this property. Because of its snapshot-based journaling, there can be two dirty copies of the same metadata block, one for the “previous” transaction being committed and the other for the current transaction.

Generation marking: The *generation marking* property requires that the file system track reuse of file pointer ob-

jects (e.g., inodes) with version numbers. Both the ext2 and ext3 file systems conform to this property; when an inode is deleted and reused for a different file, the version number of the inode is incremented. VFAT does not exhibit this property.

Delete suppression: A basic optimization found in most file systems is to suppress writes of deleted blocks. All file systems we discuss obey this property for data blocks. VFAT does not obey this property for directory blocks.

Consistent metadata: This property indicates whether the file system conveys a consistent metadata state to the storage system. All journaling file systems exhibit the consistent metadata property; transaction boundaries in their on-disk log implicitly convey this information. Ext2 and VFAT do not exhibit this property.

Data-metadata coupling: *Data-metadata coupling* builds on the consistent metadata property, and it requires the notion of consistency to be extended also to data blocks. In other words, a file system conforming to this property conveys a consistent metadata state together with the set of data blocks that were dirtied in the context of that transaction. Among the file systems we consider, only ext3 in data journaling mode conforms to this property.

5 Explicit Liveness Notification

We now proceed to the techniques for imparting various forms of liveness information to storage systems. In this section, we discuss the *explicit notification* approach, where we assume that special `allocate` and `free` commands are added to SCSI. As an optimization, we obviate the need for an explicit `allocate` command by treating a `write` to a previously freed block as an implicit `allocate`. Although modifying file systems to use this interface may seem trivial, we find that supporting the `free` command has ramifications in the consistency management of the file system under crashes.

We have modified the Linux ext2 and ext3 file systems to use this `free` command to communicate liveness information; we discuss the issues therein. The `free` command is implemented as an *ioctl* to a pseudo-device driver, which serves as our enhanced disk prototype.

5.1 Granularity of `free` notification

One issue that arises with explicit notification is the exact semantics of the `free` command, given the various granularities of liveness outlined in Section 3. For example, if only block liveness or content liveness needs to be tracked, the file system can be lazy about initiating `free` commands (thus suppressing `free` to blocks that are subsequently reused). For generation liveness, the file system needs to notify the disk of every delete of a block whose contents reached disk in the context of the deleted file. However, given multiple intermediate layers of buffering, the file system may not know exactly whether the contents of a block reached disk in the context of a certain file.

To simplify file system implementation, the file system should not be concerned about what form of liveness a particular disk functionality requires. In our approach, the file system invokes the `free` command for every logical delete. On receiving a `free` command for a block, the disk marks the block dead in its internal allocation structure (e.g., a bitmap), and on a `write`, it marks the corresponding block live. The responsibility for mapping these `free` commands to the appropriate form of liveness information lies with the disk. For example, if the disk needs to track generation deaths, it will only be interested in a `free` command to a block that it thinks is live (as indicated by its internal bitmaps); a redundant `free` to a block that is already free within the disk (which happens if the block is deleted before being written to disk) will not be viewed as a generation death. For correct operation, the file system should guarantee that it will not write a block to disk without a prior allocation; if the `write` itself is treated as an implicit `allocate`, this guarantee is the same as the *delete suppression* property. A `write` to a freed block without an allocation will result in incorrect conclusion of generation liveness within the disk. Note that after a `free` is issued for a block, the disk can safely use that block, possibly erasing its contents.

5.2 Timeliness of `free` notification

Another important issue that arises in explicit notification of a `free` is *when* the file system issues the notification. One option is *immediate notification*, where the file system issues a “free” immediately when a block gets deleted in memory. Unfortunately, this solution can result in loss of data integrity in certain crash scenarios. For example, if a crash occurs immediately after the `free` notification for a block *B* but before the metadata indicating the corresponding delete reaches disk, the disk considers block *B* as dead, while upon recovery the file system views block *B* as live since the delete never reached disk. Since a live file now contains a freed block, this scenario is a violation of data integrity. While such violations are acceptable in file systems such as ext2 which already have weak data integrity guarantees, file systems that preserve data integrity (such as ext3) need to *delay* notification until the effect of the delete reaches disk.

Delayed notification requires the file system to conform to the *reuse ordering* property; otherwise, if the block is reused (and becomes live within the file system) before the effect of the previous delete reaches disk, the delayed `free` command would need to be suppressed, which means the disk would miss a generation death.

5.3 Orphan allocations

Finally, explicit notification needs to handle the case of *orphan allocations*, where the file system considers a block dead while the disk considers it live. Assume that a block is newly allocated to a file and is written to disk in the con-

text of that file. If a crash occurs at this point (but before the metadata indicating the allocation is written to disk), the disk would assume that the block is live, but on restart, the file system views the block as dead. Since the on-disk contents of the block belong to a file that is no longer extant in the file system, the block has suffered a generation death, but the disk does not know of this. The `free` notification mechanism should enable accurate tracking of liveness despite orphan allocations. Handling orphan allocations is file system specific, as we describe below.

5.4 Explicit notification in ext2

As mentioned above, because ext2 does not provide data integrity guarantees on a crash, the notification of deletes can be immediate; thus ext2 invokes the `free` command synchronously whenever a block is freed in memory. Dealing with orphan allocations in ext2 requires a relatively simple but expensive operation; upon recovery, the `fsck` utility conservatively issues `free` notifications to every block that is currently dead within the file system.

5.5 Explicit notification in ext3

Because ext3 guarantees data integrity in its ordered and data journaling modes, `free` notification in ext3 has to be delayed until the effect of the corresponding delete reaches disk. In other words, the notification has to be delayed until the transaction that performed the delete commits. Therefore, we record an in-memory list of blocks that were deleted as part of a transaction, and issue `free` notifications for all those blocks when the transaction commits. Since ext3 already conforms to the reuse ordering property, such delayed notification is feasible.

However, a crash could occur during the invocation of the `free` commands (*i.e.*, immediately after the commit of the transaction); therefore, these `free` operations should be redo-able on recovery. For this purpose, we also log special `free` records in the journal which are then replayed on recovery, as part of the delete transaction.

During recovery, since there can be multiple committed transactions which will need to be propagated to their on-disk locations, a block deleted in a transaction could have been reallocated in a subsequent committed transaction. Thus, we cannot replay all logged `free` commands. Given our guarantee of completing all `free` commands for a transaction before committing the next transaction, we should only replay `free` commands for the last successfully committed transaction in the log (and not for any earlier committed transactions that are replayed).

To deal with orphan allocations, we log block numbers of data blocks that are about to be written, before they are actually written to disk. On recovery, ext3 can issue `free` commands to the set of orphan data blocks that were part of the uncommitted transaction.

6 Implicit Liveness Detection

In this section, we analyze various issues in implicit detection of liveness from within the storage system. Implicit liveness inference requires the storage system to have semantic understanding [24] of the on-disk format of the file system running above, coupled with careful observation of file system traffic. Because implicit liveness detection is file system dependent, we discuss the feasibility and generality of implicit liveness detection by considering three different file systems: ext2, ext3, and VFAT. In Section 8, we discuss our initial experience with implicit detection underneath the Windows NTFS file system.

Among the different forms of liveness we address, we only consider the granularity and accuracy axes mentioned in Section 3. Along the accuracy axis, we consider *accurate* and *approximate* inferences; the *approximate* instance refers to a strict *over-estimate* of the set of live entities. On the timeliness axis, we address the more common (and complex) case of lack of timely information; under most modern file systems that delay metadata updates, timeliness is not guaranteed. With guarantees of timeliness (*e.g.*, under a synchronously mounted file system), implicit inference of liveness is trivial [24].

6.1 Content liveness

As discussed in Section 3, when the disk observes a write of new contents to a live data block, it can infer that the previous contents stored in that block has suffered a content death. However, to be completely accurate, content liveness inference requires information on block liveness.

6.2 Block liveness

Block liveness information enables a storage system to know whether a given block contains valid data at any given time. To track block liveness, the storage system monitors updates to structures tracking allocation. In ext2 and ext3, there are specific data bitmap blocks which convey this information; in VFAT this information is embedded in the FAT itself, as each entry in the FAT indicates whether or not the corresponding block is free. Thus, when the file system writes an allocation structure, the storage system examines each entry and concludes that the relevant block is either dead or live.

Because allocation bitmaps are buffered in the file system and written out periodically, the liveness information that the storage system has is often stale, and does not account for new allocations (or deletes) that occurred during the interval. Table 3 depicts a time line of operations which leads to an incorrect inference by the storage system. The bitmap block M_B tracking the liveness of B is written in the first step indicating B is dead. Subsequently, B is allocated to a new file I_1 and written to disk while M_B (now indicating B as live) is still buffered in memory. At this point, the disk wrongly believes that B is dead while the on-disk contents of B are actually valid.

Operation	In-memory	On-disk
Initial	$M_B \Rightarrow B$ free	
M_B write to disk		B free
I_1 alloc	$I_1 \rightarrow B$	
	$M_B \Rightarrow B$ alloc	
B write to disk		B written
Liveness belief	B live	B free

Table 3: **Naive block liveness detection.** The table depicts a time line of events that leads to an incorrect liveness inference. This problem is solved by the shadow bitmap technique.

To address this inaccuracy, the disk tracks a *shadow copy* of the bitmaps internally [23]; whenever the file system writes a bitmap block, the disk updates its shadow copy with the copy written. In addition, whenever a data block is written to disk, the disk pro-actively sets the corresponding bit in its shadow bitmap copy to indicate that the block is live. In the above example, the write of B leads the disk to believe that B is live, thus preventing the incorrect conclusion from being drawn.

6.2.1 File system properties for block liveness

The shadow bitmap technique tracks block liveness accurately only underneath file systems that obey either the block exclusivity or data-metadata coupling property.

Block exclusivity guarantees that when a bitmap block is written, it reflects the current liveness state of the relevant blocks. If the file system tracks multiple snapshots of the bitmap block (e.g., ext3), it could write an old version of a bitmap block M_B (indicating B is dead) after a subsequent allocation and write of B . The disk would thus wrongly infer that B is dead while in fact the on-disk contents of B are valid, since it belongs to a newer snapshot; such uncertainty complicates block liveness inference.

If the file system does not exhibit block exclusivity, block liveness tracking requires the file system to exhibit data-metadata coupling, i.e., to group metadata blocks (e.g., bitmaps) with the actual data block contents in a single consistent group; file systems typically enforce such consistent groups through transactions. By observing transaction boundaries, the disk can then reacquire the temporal information that was lost due to lack of block exclusivity. For example, in ext3 data journaling mode, a transaction would contain the newly allocated data blocks together with the bitmap blocks indicating the allocation as part of one consistent group. Thus, at the commit point, the disk conclusively infers liveness state from the state of the bitmap blocks in that transaction. Since data writes to the actual in-place locations occur only after the corresponding transaction commits, the disk is guaranteed that until the next transaction commit, all blocks marked dead in the previous transaction will remain dead. In the absence of data-metadata coupling, a newly allocated data block could reach its in-place location before the corresponding transaction commits, and thus will become live in the disk before the disk detects it.

Operation	In-memory	On-disk
Initial	$M_B \Rightarrow B$ alloc	B live
	$I_1 \rightarrow B$	$I_1 \rightarrow B$
B write to disk		B written
I_1 delete	$M_B \Rightarrow B$ free	
I_2 alloc	$I_2 \rightarrow B$	
	$M_B \Rightarrow B$ alloc	
M_B write to disk		B live
Liveness belief		(Missed gen. death)

Table 4: **Missed generation death under block liveness.** The table shows a scenario to illustrate that simply tracking block liveness is insufficient to track generation deaths.

For accuracy, block liveness also requires the file system to conform to the delete suppression property; if delete suppression does not hold, a write of a block does not imply that the file system views the block as live, and thus the shadow bitmap technique will overestimate the set of live blocks until the next bitmap write. From Table 2, ext2, VFAT, and ext3 in data journaling mode thus readily facilitate block liveness detection.

6.3 Generation liveness

Generation liveness is a stronger form of liveness than block liveness, and hence builds upon the same shadow bitmap technique. With generation liveness, the goal is to find, for each on-disk block, whether a particular “generation” of data (e.g., that corresponding to a particular file) stored in that block is dead. Thus, block liveness is a special case of generation liveness; a block is dead if the latest generation that was stored in it is dead. Conversely, block liveness information is not sufficient to detect generation liveness because a block currently live could have stored a dead generation in the past. Table 4 depicts this case. Block B initially stores a generation of inode I_1 , and the disk thinks that block B is live. I_1 is then deleted, freeing up B , and B is immediately reallocated to a different file I_2 . When M_B is written the next time, B continues to be marked live. Thus, the disk missed the generation death of B that occurred between these two bitmap writes.

6.3.1 Generation liveness under reuse ordering

Although tracking generation liveness is in general more challenging, a file system that follows the reuse ordering property makes it simple to track. With reuse ordering, before a block is reused in a different file, the deleted status of the block reaches disk. In the above example, before B is reused in I_2 , the bitmap block M_B will be written, and thus the disk can detect that B is dead. In the presence of reuse ordering, tracking block liveness accurately implies accurate tracking of generation liveness. File systems such as ext3 that conform to reuse ordering, thus facilitate *accurate* tracking of generation liveness.

6.3.2 Generation liveness without reuse ordering

Underneath file systems such as ext2 or VFAT that do not exhibit the reuse ordering property, tracking generation

liveness requires the disk to look for more detailed information. Specifically, the disk needs to monitor writes to metadata objects that link blocks together into a single logical file (such as the inode and indirect blocks in ext2, the directory and FAT entries in VFAT). The disk needs to explicitly track the “generation” a block belongs to. For example, when an inode is written, the disk records that the block pointers belong to the specific inode.

With this extra knowledge about the file to which each block belongs, the disk can identify generation deaths by looking for *changes in ownership*. For example, in Table 4, if the disk tracked that B belongs to I_1 , then eventually when I_2 is written, the disk will observe a change of ownership, because I_2 owns a block that I_1 owned in the past; the disk can thus conclude that a generation death must have occurred in between.

A further complication arises when instead of being reused in I_2 , B is reused again in I_1 , now representing a new file. Again, since B now belongs to a new generation of I_1 , this scenario has to be detected as a generation death, but the ownership change monitor would miss it. To detect this case, we require the file system to track reuse of inodes (*i.e.*, the generation marking property). Ext2 already maintains such a version number, and thus enables detection of these cases of generation deaths. With version numbers, the disk now tracks for each block the “generation” it belonged to (the generation number is a combination of the inode number and the version number). When the disk then observes an inode written with an incremented version number, it concludes that all blocks that belonged to the previous version of the inode should have incurred a generation death. We call this technique *generation change monitoring*.

Finally, it is pertinent to note that the generation liveness detection through generation change monitoring is only *approximate*. Let us assume that the disk observes that block B belongs to generation G_1 , and at a later time observes that B belongs to a different generation G_2 . Through generation change monitoring, the disk can conclude that there was a generation death of B that occurred in between. However, the disk cannot know exactly *how many* generation deaths occurred in the relevant period. For example, after being freed from G_1 , B could have been allocated to G_3 , freed from G_3 and then reallocated to G_2 , but the disk never saw G_3 owning B due to delayed write of G_3 . However, as we show in our case study, this weaker form of generation liveness is still quite useful.

A summary of the file system properties required for various forms of implicit liveness inference is presented in Table 5.

7 Case Study: Secure Delete

To demonstrate our techniques for imparting liveness to storage, we present the design, implementation, and evaluation of a *secure deleting disk* under both explicit and im-

Liveness type	Properties
Block _{Approx}	Block exclusivity or Data-metadata coupling
Block _{Accurate}	[Block _{Approx}] + Delete suppression
Generation _{Approx}	[Block _{Approx}] + Generation marking
Generation _{Accurate}	[Block _{Accurate}] + Reuse ordering

Table 5: FS properties for implicit liveness detection. *Approx* indicates the set of live entities is over-estimated.

PLICIT approaches. We first describe implicit secure delete in detail, and then briefly discuss explicit secure delete.

There are two primary reasons why we chose secure deletion as our case study. First, secure delete requires tracking of generation liveness, which is the most challenging to track. Second, secure delete uses the liveness information in a context where correctness is paramount. A false positive in detecting a delete would lead to irrevocable deletion of valid data, while a false negative would result in the long-term recoverability of deleted data (a violation of secure deletion guarantees). Compared to previous work [24] which functioned only under a simplistic assumption of a synchronously mounted file system, we demonstrate that accurate inference of liveness is feasible underneath a variety of modern file system behaviors.

Our implicit secure deletion prototype is called FADED (A File-Aware Data-Erasing Disk); FADED works underneath three different file systems: ext2, VFAT, and ext3. Because of its complete lack of ordering guarantees, ext2 presented the most challenges. Specifically, since ext2 does not have the reuse ordering property, detecting generation liveness requires tracking generation information within the disk, as described in Section 6.3. We therefore mainly focus on the implementation of FADED underneath ext2, and finally discuss some key differences in our implementation for other file systems.

7.1 Goals of FADED

The desired behavior of FADED is as follows: for every block that reaches the disk in the context of a certain file F , the delete of file F should trigger a secure overwrite (*i.e.*, *shred*) of the block. This behavior corresponds to the notion of *generation liveness* defined in Section 3. A *shred* involves multiple overwrites to the block with specific patterns so as to erase remnant magnetic effects of past layers (that could otherwise be recovered through techniques such as magnetic scanning tunneling microscopy [12]). Recent work suggests that two such overwrites are sufficient to ensure non-recoverability in modern disks [14].

Traditionally, secure deletion is implemented within the file system [3, 25, 26]; however, such implementations are unreliable given modern storage systems. First, for high security, overwrites need to be *off-track* writes (*i.e.*, writes straggling physical track boundaries), which external erase programs (*e.g.*, the file system) cannot perform [13]. Further, if the storage system buffers writes in NVRAM [32], multiple overwrites done by the file system

may be collapsed into a single write to the physical disk, making the overwrites ineffective. Finally, in the presence of block migration [7, 32] within the storage system, an overwrite by the file system will only overwrite the current block location; stray copies of deleted data could remain. Thus, the storage system is the proper locale to implement secure deletion.

Note that FADED operates at the granularity of an entire volume; there is no control over which individual files are shredded. However, this limitation can be dealt with by storing “sensitive” files in a separate volume on which the secure delete functionality is enabled.

7.2 Basic operation

As discussed in Section 6.3, FADED monitors writes to inode and indirect blocks and tracks the inode generation to which each block belongs. It augments this information with the block liveness information it collects through the shadow bitmap technique. Note that since ext2 obeys the block exclusivity and delete suppression properties, block liveness detection is reliable. Thus, when a block death is detected, FADED can safely shred that block.

On the other hand, if FADED detects a generation death through the ownership change or generation change monitors (*i.e.*, the block is live according to the block liveness module), FADED cannot simply shred the block, because FADED does not know if the current contents of the block belong to the generation that was deleted, or to a new generation that was subsequently allocated the same block due to block reuse. If the current contents of the block are valid, a shredding of the block would be catastrophic.

We deal with such uncertainty through a conservative approach to generation-death inference. By being conservative, we convert an apparent correctness problem into a performance problem, *i.e.*, we may end up performing more overwrites than required. Fundamental to this approach is the notion of a *conservative overwrite*.

7.2.1 Conservative overwrites

A conservative overwrite of block *B* erases past layers of data on the block, but leaves the current contents of *B* intact. Thus, even if FADED does not know whether a subsequent valid write occurred after a predicted generation death, a conservative overwrite on block *B* will be safe; it can never shred valid data. To perform a conservative overwrite of block *B*, FADED reads the block *B* into non-volatile RAM, then performs a normal secure overwrite of the block with the specific pattern, and ultimately restores the original data back into block *B*.

The problem with a conservative overwrite is that if the block contents that are restored after the conservative overwrite are in fact the old data (which had to be shredded), the conservative overwrite was ineffective. In this case, FADED can be guaranteed to observe one of two things. First, if the block had been reused by the file sys-

tem for another file, the new, valid data will be written eventually (*i.e.*, within the delayed write interval of the file system). When FADED receives this new write, it buffers the write, and before writing the new data to disk, FADED performs a shred of the concerned block once again; this time, FADED knows that it need not restore the old data, because it has the more recent contents of the block. To identify which writes to treat in this special manner, FADED tracks the list of blocks that were subjected to a conservative overwrite in a *suspicious blocks* list, and a write to a block in this list will be committed only after a secure overwrite of the block; after the second overwrite, the block is removed from the suspicious list. Note that the suspicious list needs to be stored persistently, perhaps in NVRAM, in order to survive crashes.

Second, if the block is not reused by the file system immediately, then FADED is guaranteed to observe a bitmap reset for the corresponding block, which will be flagged as a block death by the block liveness detector. Since block liveness tracking is reliable, FADED can now shred the block again, destroying the old data. Thus, in both cases of wrongful restore of old data, FADED is guaranteed to get another opportunity to make up for the error.

7.2.2 Cost of conservatism

Conservative overwrites come with a performance cost; every conservative overwrite results in the concerned block being treated as “suspicious”, regardless of whether the data restored after the conservative overwrite was the old or new data, because FADED has no information to find it at that stage. Because of this uncertainty, even if the data restored were the new data (and hence need not be overwritten again), a subsequent write of the block in the context of the same file would lead to a redundant shredding of the block. Here we see one example of the performance cost FADED pays to circumvent the lack of perfect information.

7.3 Coverage of deletes

In the previous subsection, we showed that for all generation deaths detected, FADED ensures that the appropriate block version is overwritten, without compromising valid data. However, for FADED to achieve its goals, these detection techniques must be *sufficient* to identify *all* cases of deletes at the file system level that need to be shredded. In this section, we show that FADED can indeed detect all deletes, but requires two minor modifications to ext2.

7.3.1 Undetectable deletes

Because of the weak properties of ext2, certain deletes can be missed by FADED. We present the two specific situations where identification of deletes is impossible, and then propose minor changes to ext2 to fix those scenarios.

File truncates: The generation change monitor assumes that the version number of the inode is incremented when the inode is reused. However, the version number in ext2

Operation	In-memory	On-disk
Initial	$I_1 \rightarrow B^{Ind}$	$I_1 \rightarrow B^{Ind}$
I_1 delete	B free	
I_2 alloc	$I_2 \rightarrow B$	
B write to disk		$I_1 \rightarrow B^{Ind}$ (wrong type)

Table 6: **Misclassified indirect block.** The table shows a scenario where a normal data block is misclassified as an indirect block. B^{Ind} indicates that B is treated as an indirect block. Reuse ordering for indirect blocks prevents this problem.

is only incremented on a complete delete and reuse; partial truncates do not affect the version number. Thus if a block is freed due to a partial truncate and is reassigned to the same file, FADED misses the generation death. Although such a reuse after a partial truncate could be argued as a logical overwrite of the file (and thus, not a *delete*), we adopt the more complex (and conservative) interpretation of treating it as a delete.

To handle such deletes, we propose a small change to ext2; instead of incrementing the version number on a re-allocation of the inode, we increment it on every truncate. Alternatively, we could introduce a separate field to the inode that tracks this version information. This is a non-intrusive change, but is effective at providing the disk with the requisite information. This technique could result in extra overwrites in the rare case of partial truncates, but correctness is guaranteed because the “spurious” overwrites would be conservative and would leave data intact.

Reuse of indirect blocks: A more subtle problem arises due to the presence of indirect pointer blocks. Indirect blocks share the data region of the file system with other user data blocks; thus the file system can reuse a normal user data block as an indirect block and vice versa. In the presence of such *dynamic typing*, the disk cannot reliably identify an indirect block [23].

The only way FADED can identify a block B as an indirect block is when it observes an inode I_1 that contains B in its indirect pointer field. FADED then records the fact that B is an indirect block. However, when it later observes a write to B , FADED cannot be certain that the contents indeed are those of the indirect block, because in the meanwhile I_1 could have been deleted, and B could have been reused as a user data block in a different inode I_2 . This scenario is illustrated in Table 6.

Thus, FADED cannot trust the block pointers in a suspected indirect block; this uncertainty can lead to missed deletes in certain cases. To prevent this occurrence, a data block should never be misclassified as an indirect block. To ensure this, before the file system allocates, and immediately after the file system frees an indirect block B^{Ind} , the concerned data bitmap block $M_{B^{Ind}}$ should be flushed to disk, so that the disk will know that the block was freed. Note that this is a weak form of reuse ordering only for indirect blocks. As we show later, this change has very little

Operation	In-memory	On-disk
Initial	B free	B free
I_1 alloc	$I_1 \rightarrow B$	
B write to disk		B written
I_1 delete	B free	
I_2 alloc	$I_2 \rightarrow B$	
I_2 write to disk		$I_2 \rightarrow B$ (Missed delete of B)

Table 7: **Missed delete due to an orphan write.** The table illustrates how a delete can be missed if an orphan block is not treated carefully. Block B , initially free, is allocated to I_1 in memory. Before I_1 is written to disk, B is written. I_1 is then deleted and B reallocated to I_2 . When I_2 is written, FADED would associate B with I_2 and would miss the overwrite of B .

impact on performance, since indirect blocks tend to be a very small fraction of the set of data blocks.

Practicality of the changes: The two changes discussed above are minimal and non-intrusive; the changes together required modification of 12 lines of code in ext2. Moreover, they are required only because of the weak ordering guarantees of ext2. In file systems such as ext3 which exhibit reuse ordering, these changes are not required. Our study of ext2 is aimed as a limit study of the minimal set of file system properties required to reliably implement secure deletion at the disk.

7.3.2 Orphan allocations

Implicit block liveness tracking in FADED already addresses the orphan allocation issue discussed in § 5.3; when ext2 recovers after a crash, the fsck utility writes out a copy of all bitmap blocks; the block liveness monitor in FADED will thus detect death of those orphan allocations.

7.3.3 Orphan writes

Due to arbitrary ordering in ext2, FADED can observe a write to a newly allocated data block before it observes the corresponding owning inode. Such *orphan writes* need to be treated carefully because if the owning inode is deleted before being written to disk, FADED will never know that the block once belonged to that inode. If the block is reused in another inode, FADED would miss overwriting the concerned block which was written in the context of the old inode. Table 7 depicts such a scenario.

One way to address this problem is to *defer* orphan block writes until FADED observes an owning inode [23], a potentially memory-intensive solution. Instead, we use the suspicious block list used in conservative overwrites to also track orphan blocks. When FADED observes a write to an orphan block B , it marks B suspicious; when a subsequent write arrives to B , the old contents are shredded. Thus, if the inode owning the block is deleted before reaching disk, the next write of the block in the context of the new file will trigger the shred. If the block is not reused, the bitmap reset will indicate the delete.

This technique results in a redundant secure overwrite anytime an orphaned block is overwritten by the file sys-

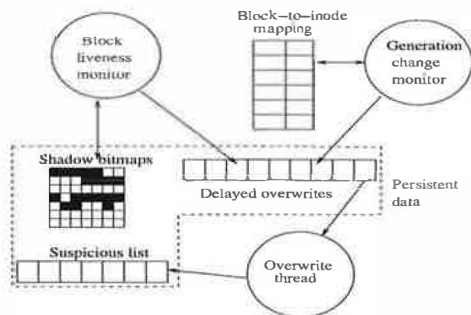


Figure 1: Key components of FADED.

tem in the context of the same file, again a cost we pay for conservatism. Note that this overhead is incurred only the first time an orphan block is overwritten.

7.3.4 Guaranteed detection of deletes

With these techniques, we can prove that for every block *B* that is deleted by the file system after it has reached disk, FADED always overwrites the deleted contents of *B*. The proof is presented in Appendix A.

7.4 Delayed overwrites

Multiple overwrites of the same block cause additional disk I/Os that can hurt performance if incurred on the critical path. For better performance, FADED delays overwrites until idle time in the workload [11] (or optionally, until up to *n* minutes of detection). Thus, whenever FADED decides to shred a block, it just queues it; a low priority thread services this queue if FADED had not observed useful foreground traffic for more than a certain duration. Delayed overwrites help FADED to present writes to the disk in a better, sequential ordering, besides reducing the impact on foreground performance. Delaying also reduces the number of overwrites if the same block is deleted multiple times. The notion of conservative overwrites is crucial to delaying overwrites arbitrarily, even after the block that had to be overwritten is written in the context of a new file. Note that if immediate shredding is required, the user needs to perform a `sync`.

A summary of the key data structures and components of FADED is presented in Figure 1.

7.5 FADED for other file systems

We have also implemented FADED underneath other file systems, and in each case, validated our implementation with the same testing methodology as will be described in Section 7.7. However, due to space constraints, we only point to the key differences we observed relative to ext2.

7.5.1 FADED for VFAT

Like ext2, VFAT also does not conform to reuse ordering, so FADED needs to track generation information for each block in order to detect deletes. One key difference in VFAT compared to ext2 is that there are no pre-allocated, uniquely addressable “inodes”, and consequently, no “version” information as well. Dynamically

allocated directory blocks contain a pointer to the start block of a file; the FAT chains the start block to the other blocks of the file. Thus, detecting deletes reliably underneath unmodified VFAT is impossible. We therefore introduced an additional field to a VFAT directory entry that tracks a globally unique *generation number*. The generation number gets incremented on every create and delete in the file system, and a newly created file is assigned the current value of generation number. With this small change (29 lines of code) to VFAT, the generation change monitor accurately detects all deletes of interest.

7.5.2 FADED for ext3

Since ext3 exhibits reuse ordering, tracking generation liveness in ext3 is the same as tracking block liveness. However, since ext3 does not obey the block exclusivity property, tracking block liveness accurately is impossible except in the data journaling mode which has the useful property of data-metadata coupling. For the ordered and writeback modes, we had to make a small change: when a metadata transaction is logged, we also made ext3 log a list of data blocks that were allocated in the transaction. This change (95 lines of code), coupled with the reuse ordering property, enables accurate tracking of deletes.

7.6 Explicit secure delete

We have also built secure deletion under the explicit notification framework. We modified the ext2 and ext3 file systems to notify the disk of every logical delete (as described in §5). The file system modifications accounted for 14 and 260 lines of code respectively. Upon receiving the notification, the disk decides to shred the block. However, similar to FADED, the disk delays overwrites until idle time to minimize impact on foreground performance.

7.7 Evaluation

In this section, we evaluate our implicit and explicit implementations of secure delete. The enhanced disk is implemented as a pseudo-device driver in the Linux 2.4 kernel; the driver observes the same information as a hardware prototype, but suffers contention for CPU and memory from the host. We use a 2.4 GHz Pentium-4 with 1 GB RAM and a 10K RPM IBM 9LZX disk. Due to space constraints, we provide results only for the ext2 version.

7.7.1 Correctness and accuracy

To test whether our FADED implementation detected all deletes of interest, we instrument the file system to log every delete, and correlate it with the log of writes and overwrites by FADED, to capture cases of unnecessary or missed overwrites. We tested our system on various workloads with this technique, including a few busy hours from the HP file system traces [19]. Table 8 presents the results of this study on the trace hour 09 00 of 11/30/00.

In this experiment, we ran FADED under four versions of Linux ext2. In the first, marked “No changes”, a default

Config	Delete	Overwrite	Excess	Miss
No changes	76948	68700	11393	854
Indirect	76948	68289	10414	28
Version	76948	69560	11820	0
Both	76948	67826	9610	0

Table 8: **Correctness and accuracy.** The table shows the number of overwrites performed by the FADED under various configurations of ext2. The columns (in order) indicate the number of blocks deleted within the file system, the total number of logical overwrites performed by FADED, the number of unnecessary overwrites, and the number of overwrites missed by FADED. Note that deletes that occurred before the corresponding data write do not require an overwrite.

Config	Reads	Writes	Run-time(s)
No changes	394971	234664	195.0
Version	394931	234648	195.5
Both	394899	235031	200.0

Table 9: **Impact of FS changes on performance.** The performance of the various file system configurations under a busy hour of the HP Trace is shown. For each configuration, we show the number of blocks read and written, and the trace run-time.

ext2 file system was used. In “Indirect”, we used ext2 modified to obey reuse ordering for indirect blocks. In “Version”, we used ext2 modified to increment the inode version number on every truncate, and the “Both” configuration represents both changes (the correct file system implementation required for FADED). The third column gives a measure of the extra work FADED does in order to cope with inaccurate information. The last column indicates the number of missed overwrites; in a correct system, the fourth column should be zero.

We can see that the cost of inaccuracy is quite reasonable; FADED performs roughly 14% more overwrites than the minimal amount. Also note that without the version number modification to ext2, FADED indeed misses a few deletes. The reason no missed overwrites are reported for the “Version” configuration is the rarity of the case involving a misclassified indirect block.

7.7.2 Performance impact of FS changes

We next evaluate the performance impact of the two changes we made to ext2, by running the same HP trace on different versions of ext2. Table 9 shows the results. As can be seen, even with both changes, the performance reduction is only about 2% and the number of blocks written is marginally higher due to synchronous bitmap writes for indirect block reuse ordering. We thus conclude that the changes are quite practical.

7.7.3 Performance of secure delete

We now explore the foreground performance of implicit and explicit secure delete, and the cost of overwrites.

Foreground performance impact: Tracking block and generation liveness requires FADED to perform extra processing. This cost of reverse engineering directly impacts application performance because it is incurred on the crit-

System	Run-time (s)		
	PostMark	HP Trace	Explicit HP Trace
Default	166.8	200.0	195.0
SecureDelete ₂	177.7	209.6	195.5
SecureDelete ₄	178.4	209.0	196.8
SecureDelete ₆	179.0	209.3	196.4

Table 10: **Foreground impact: Postmark and HP trace.** The run-times for Postmark and the HP trace are shown for FADED, with 2, 4 and 6 overwrite passes. For comparison, the run-time of explicit secure delete on the HP Trace is also shown. Postmark was configured with 40K files and 40K transactions.

ical path of every disk operation. We quantify the impact of this extra processing required at FADED on foreground performance. Since our software prototype competes for CPU and memory resources with the host, these are worst case estimates of the overheads.

We run the Postmark file system benchmark [15] and the HP trace on a file system running on top of FADED. Postmark is a metadata intensive small-file benchmark, and thus heavily exercises the inferencing mechanisms of FADED. To arrive at a pessimistic estimate, we perform a sync at the end of each phase of Postmark, causing all disk writes to complete and account that time in our results. Note that we do not wait for completion of delayed overwrites. Thus, the numbers indicate the performance perceived by the foreground task.

Table 10 compares the performance of FADED both with a default disk and with explicit secure delete. From the table, we can see that even for 4 or 6 overwrite passes, foreground performance is not affected much. Extra CPU processing within FADED causes only about 4 to 7% lower performance compared to the modified file system running on a normal disk. The explicit implementation performs better because it does not incur the overhead of inference. Further, it does not require the file system modifications reported in Table 9 (this corresponds to the “No changes” row in Table 9). Note that we do not model the cost of sending a free command across the SCSI bus; thus the overheads in the explicit case are optimistic.

Idle time required: We now quantify the cost of performing overwrites for shredding. With micro-benchmarks, we verified that the overwrites obtained near sequential bandwidth due to their delayed, ordered issue. We also found that when block reuse occurs within the file system (resulting in multiple deletes to the same block), delaying overwrites significantly reduces overwrite traffic. We omit these results due to space constraints.

We next explore the time required for overwrites. First, we use the same Postmark configuration as above, but measure the time for the benchmark to complete including delayed overwrites. Since Postmark deletes all files at the end of its run, we face a worst case scenario where the entire working set of the benchmark has to be overwrit-

System	Run-time with overwrites (s)		
	Implicit PostMark	HP Trace	Explicit HP Trace
Default	166.8	200.0	195.0
SecureDelete ₂	466.6	302.8	280.0
SecureDelete ₄	626.4	345.6	316.2
SecureDelete ₆	789.3	394.3	346.1

Table 11: **Idle time requirement.** The table shows the total run-time of two benchmarks, Postmark and the HP trace. The time reported includes completion of all delayed overwrites.

ten, accounting for the large overwrite times reported in Table 11. In the HP-trace, the overwrite times are more reasonable. Since most blocks deleted in the HP trace are then reused in subsequent writes, most of the overwrites performed here are conservative. This accounts for the steep increase from 0 to 2 overwrite passes, in the implicit case. The explicit implementation incurs 8-13% lower overwrite times compared to FADED because it has perfect information on deletes, and thus avoids extra overwrites incurred due to conservatism.

8 Implicit Detection Under NTFS

In this section, we present our experience building support for implicit liveness detection underneath the Windows NTFS file system. The main challenge we faced underneath NTFS was the absence of source code for the file system. While the basic on-disk format of NTFS is known [27], details of its update semantics and journaling behavior are not publicly available. As a result, our implementation currently tracks only block liveness which requires only knowledge of the on-disk layout; generation liveness tracking could be implemented if the details of NTFS journaling mechanism were known.

The fundamental piece of metadata in NTFS is the Master File Table (MFT); each record in the MFT contains information about a unique file. Every piece of metadata in NTFS is treated as a regular file; file 0 is the MFT itself, file 2 is the recovery log, and so on. The allocation status of all blocks in the volume is maintained in a file called the cluster bitmap, which is similar to the block bitmap tracked by ext2. On block allocations and deletions, NTFS regularly writes out modified bitmap blocks.

Our prototype implementation runs as a device driver in Linux, similar to the setup described earlier for other file systems. The virtual disk on which we interpose is exported as a logical disk to a virtual machine instance of Windows XP running over VMware Workstation [30]. To track block liveness, our implementation uses the same shadow bitmap technique mentioned in Section 6.2. By detailed empirical observation under long-running workloads, we found that NTFS did not exhibit any violation of the block exclusivity and delete suppression properties mentioned in Section 4.2; however, due to the absence of source code, we cannot assert that NTFS *always* conforms to these properties. This limitation points to the

general difficulty of using implicit techniques underneath closed-source file systems; one can never be certain that the file system conforms to certain properties unless those are guaranteed by the file system vendor. In the absence of such guarantees, the utility of implicit techniques is limited to optimizations that can afford to be occasionally “wrong” in their implicit inference.

Our experience with NTFS also points to the utility of characterizing the precise set of file system properties required for various forms of liveness inference. This set of properties now constitutes a minimal “interface” for communication between file system and storage vendors. For example, if NTFS confirmed its conformance to the block exclusivity and delete suppression properties, the storage system could safely implement aggressive optimizations that rely on its implicit inference.

9 Discussion

In this section, we reflect on the lessons learned from our case study to refine our comparison on the strengths and weaknesses of the explicit and implicit approaches.

The ideal scenario for the implicit approach is where changes are required only in the storage system and not in the file system or the interface. However, in practice, accurate liveness detection requires certain file system properties, which means the file system needs to be modified if it does not conform to those requisite properties. In the face of such changes to both the storage system and the file system, it might appear that the implicit approach is not much more pragmatic than the explicit approach of changing the interface also. There are two main reasons why we believe the implicit approach is still useful.

First, file system changes are not required if the file system already conforms to the requisite properties. For example, many file systems (e.g. ext2, VFAT, ext3-data journaling, and perhaps NTFS) are already amenable to block liveness detection without any change to the file system. The ext3 file system in data journaling mode already conforms to the properties required for generation liveness detection. Clearly, in such cases, the implicit approach enables non-intrusive deployment of functionality.

Second, we believe that modifying the file system to conform to a set of well-defined properties is more general than modifying the file system (and the interface) to convey a specific piece of information. Although we have discussed the file system properties from the viewpoint of implicit liveness detection, some of the properties enable richer information to be inferred; for example, the association between a block and its owning inode (required for certain applications such as file-aware layout [23]) can be tracked accurately if the file system obeys the reuse ordering or the consistent metadata properties. Our ultimate goal is to arrive at a set of properties that enable a wide variety of information to be tracked implicitly, thus outlining how file systems may need to be designed to enable

such transparent extension within the storage system. In contrast, the approach of changing the interface requires introducing a new interface every time a different piece of information is required.

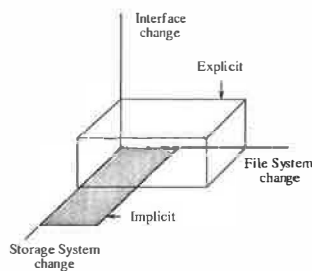
10 Related Work

The need for liveness information in storage systems has been recognized in previous work. In most existing proposals, an interface to communicate liveness is a part of a more radical set of changes to the existing storage interface. For example, logical disks have a list-based interface to storage which includes a command to “delete” a block from a list [4]. More recent work suggests an object-like interface to storage [17], which moves the responsibilities of low-level storage management such as liveness tracking from the file system into the drives themselves. In contrast to such wide-scale changes, our “explicit notification” approach for imparting liveness is much less intrusive on the large body of file systems that utilize the existing block-based interface to storage.

There has also been work on implementing “smarts” within a storage system without interface change, similar to our implicit approach. Some of these systems utilize a limited form of liveness inference. For example, AutoRAID requires information on free space to decide the amount of data that can be stored in RAID-1 [32]; AutoRAID infers that blocks that have not been written ever, are dead. This inference is a weak form of liveness because once a block is written, subsequent deletes cannot be detected. Other systems such as the programmable disk [31] make similar inferences. The existence of these proposals indicates that liveness information is important in storage systems, and yet systematic techniques for acquiring such information have been missing.

Most related to the implicit techniques in this work is our previous work on semantically-smart disks [24]. In that work, we presented techniques by which a block-based storage system can infer file system level information and implemented a set of case studies such as track-aligned extents, journaling, and secure delete. However, all correctness-sensitive case studies implemented therein required the file system to be synchronously mounted; under synchronous file systems, implicit information tracking is trivial. Our more recent work on D-GRAID [23] considered asynchronous file systems, but the layout mechanisms of D-GRAID did not depend on accuracy for correctness; it was acceptable in D-GRAID to get predictions wrong. Also, fast recovery in D-GRAID utilized *block liveness* (a much easier property to track than generation liveness) under specific assumptions on file system behavior. In this work, we go beyond our previous work by generalizing our techniques for inference underneath a wide range of realistic file system behaviors, and demonstrating that storage-level functionality where correctness is paramount, can utilize this information reliably.

11 Conclusion



As system layers evolve over time, interfaces between layers become obsolete or sub-optimal, necessitating their evolution. We have presented two approaches for interface evolution: explicit and implicit, in the context of embedding liveness information into storage.

A qualitative summary of the complexity of the two approaches along various axes is presented in the figure. We have shown that the explicit approach, while appearing straightforward, entails a fair amount of file system change in practice, besides requiring some minimal support from the storage system. Despite these factors, the explicit approach results in simpler systems than the implicit case. The main strength of the implicit approach is that it permits demonstration of functionality without changes to the interface, thus enabling seamless deployment while catalyzing rapid interface evolution.

Acknowledgments

We thank Nitin Agrawal, John Bent, Timothy Denehy, Todd Jones, James Nugent, Florentina Popovici, and Vinod Yegneswaran for their helpful comments. We also thank Mendel Rosenblum for his excellent shepherding, the anonymous reviewers for their thoughtful feedback, and Gordon Hughes for his useful comments on secure delete. This work is sponsored by NSF CCR-0092840, CCR-0133456, CCR-0098274, NGS-0103670, ITR-0086044, ITR-0325267, IBM and EMC.

References

- [1] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *28th VLDB*, 2002.
- [2] L. Bairavasundaram, M. Sivathanu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. X-RAY: A Non-Invasive Exclusive Caching Mechanism for RAIDs. In *ISCA '04*, 2004.
- [3] S. Bauer and N. B. Priyantha. Secure Data Deletion for Linux File Systems. In *USENIX Security*, August 2001.
- [4] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The Logical Disk: A New Approach to Improving File Systems. In *SOSP '93*, 1993.
- [5] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the Information Gap in Storage Protocol Stacks. In *USENIX*, Monterey, CA, June 2002.
- [6] I. Dowse and D. Malone. Recent Filesystem Optimisations on FreeBSD. In *FREENIX*, June 2002.
- [7] EMC Corporation. Symmetrix Enterprise Information Storage Systems. <http://www.emc.com>, 2002.
- [8] R. M. English and A. A. Stepanov. Loge: A Self-Organizing Disk Controller. In *USENIX*, Jan. 1992.
- [9] G. R. Ganger. Blurring the Line Between Oses and Storage Devices. TR SCS CMU-CS-01-166, Dec. 2001.
- [10] G. R. Ganger, M. K. McKusick, C. A. Soules, and Y. N. Patt. Soft Updates: A Solution to the Metadata Update Problem in File Systems. *ACM TOCS*, 18(2), May 2000.
- [11] R. A. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes. Idleness is not sloth. In *USENIX Winter*, pages 201–212, 1995.
- [12] P. Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *USENIX Security*, July 1996.
- [13] G. Hughes. Personal communication, 2004.

- [14] G. Hughes and T. Coughlin. Secure Erase of Disk Drive Data. IDEMA Insight Magazine, 2002.
- [15] J. Katcher. PostMark: A New File System Benchmark. NetApp TR-3022, October 1997.
- [16] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *TOCS*, 2(3), Aug. 1984.
- [17] M. Mesnier, G. R. Ganger, and E. Riedel. Object-Based Storage. *IEEE Communications Magazine*, 41(8), August 2003.
- [18] A. Pennington, J. Strunk, J. Griffin, C. Soules, G. Goodson, and G. Ganger. Storage-based Intrusion Detection: Watching Storage Activity For Suspicious Behavior. In *USENIX Security*, 2003.
- [19] E. Riedel, M. Kallahalla, and R. Swaminathan. A Framework for Evaluating Storage System Security. In *FAST '02*, 2002.
- [20] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. In *USENIX '00*, 2000.
- [21] C. Ruemmler and J. Wilkes. Disk Shuffling. Technical Report HPL-91-156, HP Laboratories, 1991.
- [22] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics. In *FAST '02*, January 2002.
- [23] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. In *FAST '04*, Mar. 2004.
- [24] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *FAST '03*, 2003.
- [25] SourceForge. SRM: Secure File Deletion for POSIX Systems. <http://srm.sourceforge.net>, 2003.
- [26] SourceForge. Wipe: Secure File Deletion. <http://wipe.sourceforge.net>, 2003.
- [27] SourceForge. The Linux NTFS Project. <http://linux-ntfs.sf.net/>, 2004.
- [28] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. Soules, and G. R. Ganger. Self-Securing Storage: Protecting Data in Compromised Systems. In *OSDI 2000*, 2000.
- [29] T. Ts'o and S. Tweedie. Future Directions for the Ext2/3 Filesystem. In *FREENIX*, June 2002.
- [30] VMWare. VMWare Workstation 4.5. <http://www.vmware.com/products/>, 2004.
- [31] R. Wang, T. E. Anderson, and D. A. Patterson. Virtual Log-Based File Systems for a Programmable Disk. In *OSDI '99*, 1999.
- [32] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.
- [33] X. Yu, B. Gum, Y. Chen, R. Wang, K. Li, A. Krishnamurthy, and T. Anderson. Trading Capacity for Performance in a Disk Array. In *OSDI '00*, 2000.

A Guaranteed detection of deletes

We now prove that the techniques in FADED for ext2 guarantee shredding of all deletes of blocks whose contents reached disk.

When a delete of an inode I_1 occurs within ext2, a set of blocks are freed from a file; this results in an increment of the version number of I_1 , and the reset of relevant bits in the data bitmap block pertaining to the freed blocks. Let us consider one such block B that is freed. Let us assume that B had already been written to disk in the context of I_1 . If B had not been written to disk, the disk does not need to perform any overwrite, so we do not consider that case. Let the bitmap block containing the status of B be M_B , and let B_I be the block containing the inode I_1 . Now, there are two possibilities: either B is reused by the file system before M_B is written to disk, or B is not reused until the write of M_B .

Case 1: Block B not reused

If B is not reused immediately to a different file, the bitmap block M_B , which is dirtied, will be eventually

written to disk, and the disk will immediately know of the delete through the block liveness module, and thus overwrite B .

Case 2: Block B is reused

Let us now consider the case where B is reused in inode I_2 . There are three possibilities in this case: at the point of receiving the write of B , the disk either thinks B belongs to I_1 , or it thinks B is free, or that B belongs to some other inode I_x .

Case 2a: Disk thinks $I_1 \rightarrow B$

If the disk knew that $I_1 \rightarrow B$, the disk would have tracked the previous version number of I_1 . Thus, when it eventually observes a write of B_I , (which it will, since B_I is dirtied because of the version number increment), the disk will note that the version number of I_1 has increased, and thus would overwrite all blocks that it thought belonged to I_1 , which in this case includes B . Thus B would be overwritten, perhaps restoring a newer value. As discussed in Section 7.2, even if this was a conservative overwrite, the old contents are guaranteed to be shredded.

Case 2b: Disk thinks B is free

If the disk thinks B is free, it would treat B as an orphan block when it is written, and mark it suspicious. Consequently, when B is written again in the context of the new inode I_2 , the old contents of B will be shredded.

Case 2c: Disk thinks $I_x \rightarrow B$

To believe that $I_x \rightarrow B$, the disk should have observed I_x pointing to B at some point before the current write to B .¹ The disk could have observed $I_x \rightarrow B$ either before or after B was allocated to I_1 by the file system.

Case 2c-i: $I_x \rightarrow B$ before $I_1 \rightarrow B$

If the disk observed $I_x \rightarrow B$ before it was allocated to I_1 , and still thinks $I_x \rightarrow B$ when B is written in the context of I_1 , it means the disk never saw $I_1 \rightarrow B$. However, in this case, block B was clearly deleted from I_x at some time in the past in order to be allocated to I_1 . This would have led to the version number of I_x incrementing, and thus when the disk observes I_x written again, it would perform an overwrite of B since it thinks B used to belong to I_x .

Case 2c-ii: $I_x \rightarrow B$ after $I_1 \rightarrow B$

If this occurs, it means I_x was written to disk owning B after B was deleted from I_1 but before B is written. In this case, B will only be written in the context of I_x which is still live, so it does not have to be overwritten. As discussed in Section 4.2, this holds because of the block exclusivity property of ext2.

Note that the case of a block being deleted from a file and then quickly reallocated to the same file is just a special case of Case 2c, with $I_1 = I_x$.

Thus, in all cases where a block was written to disk in the context of a certain file, the delete of the block from the file will lead to a shred of the deleted contents. \square

¹If indirect block detection was uncertain, the disk can wrongly think $I_x \rightarrow B$ because of a corrupt "pointer" in a false indirect block; our file system change for reuse ordering in indirect blocks prevents this case.

Program-Counter-Based Pattern Classification in Buffer Caching

Chris Gniady Ali R. Butt Y. Charlie Hu

Purdue University

West Lafayette, IN 47907

{gniady, butta, ychu}@purdue.edu

Abstract

Program-counter-based (PC-based) prediction techniques have been shown to be highly effective and are widely used in computer architecture design. In this paper, we explore the opportunity and viability of applying PC-based prediction to operating systems design, in particular, to optimize buffer caching. We propose a Program-Counter-based Classification (PCC) technique for use in pattern-based buffer caching that allows the operating system to correlate the I/O operations with the program context in which they are issued via the program counters of the call instructions that trigger the I/O requests. This correlation allows the operating system to classify I/O access pattern on a per-PC basis which achieves significantly better accuracy than previous per-file or per-application classification techniques. PCC also performs classification more quickly as per-PC pattern just needs to be learned once. We evaluate PCC via trace-driven simulations and an implementation in Linux, and compare it to UBM, a state-of-the-art pattern-based buffer replacement scheme. The performance improvements are substantial: the hit ratio improves by as much as 29.3% (with an average of 13.8%), and the execution time is reduced by as much as 29.0% (with an average of 13.7%).

1 Introduction

One of the most effective optimization techniques in computer systems design is history-based prediction, based on the principle that most programs exhibit certain degrees of repetitive behavior. Such history-based prediction techniques include cache replacement and prefetching schemes for various caches inside a computer system such as hardware memory caches, TLBs, and buffer caches inside the operating system, as well as processor oriented optimizations such as branch prediction.

A key observation in the processor architecture is that program instructions (or their program counters) provide

a highly effective means of recording the context of program behavior. Consequently, program-counter-based (PC-based) prediction techniques have been extensively studied and widely used in modern computer architectures. However, despite their tremendous success in architecture design, PC-based techniques have not been explored in operating systems design.

In this paper, we explore the opportunities and feasibility of PC-based techniques in operating systems design. In particular, we consider the application of PC-based prediction to the I/O management in operating systems. Since the main memory, like hardware caches, is just another level of the memory hierarchy, the I/O behavior of a program is expected to be similar to the data movement in the upper levels of the memory hierarchy in that there is a strong correlation between the I/O behavior and the program context in which the I/O operations are triggered. Furthermore, program counters are expected to remain as an effective means of recording such program context, similar to recording the context of data movement in the upper levels of memory hierarchy.

The specific PC-based prediction technique we propose, called Program-Counter-based Classification (PCC), identifies the access pattern among the blocks accessed by I/O operations triggered by a call instruction in the application. Such pattern classifications are then used by a pattern-based buffer cache to predict the access patterns of blocks accessed in the future by the same call instruction. A suitable replacement algorithm is then used to manage the accessed disk blocks belonging to each pattern as in previous pattern-based buffer cache schemes. Thus, PCC correlates the I/O access patterns observed by the operating system with the program context in which they are issued, i.e., via the PC of the call instruction that triggers the I/O requests. This correlation allows the operating system to classify block accesses on a *per-PC basis*, and distinguishes PCC from previous classification schemes in two fundamental aspects. First, if the same instruction is observed again for newly opened files, PCC

can immediately predict the access pattern based on history. Second, PCC can differentiate multiple reference patterns in the same file if they are invoked by different instructions.

The design of PCC faces several challenges. First, retrieving the relevant program counter that is responsible for triggering an I/O operation can be tricky, as I/O operations in application programs are often abstracted into multiple, layered subroutines and different call sites may go through multiple wrappers before invoking some shared I/O system calls. Second, PCC requires a predefined set of access patterns and implements their detection algorithms. Third, the buffer cache needs to be partitioned into subcaches, one for each pattern, and the sub-cache sizes need to be dynamically adjusted according to the distribution of the blocks of different patterns.

This paper makes the following contributions.

- It is, to our knowledge, the first to apply a PC-based prediction in operating systems design;
- It presents the first technique that allows the operating system to correlate I/O operations with the program context in which they are triggered;
- It presents experimental results demonstrating that correlating I/O operations with the program context allows for a more accurate and adaptive prediction of I/O access patterns than previous classification schemes, and a pattern-based buffer caching scheme using PCC outperforms state-of-the-art recency/frequency-based schemes;
- It shows that exploiting the synergy between architecture and operating system techniques to solve problems of common characteristics, such as exploiting the memory hierarchy, is a promising research direction.

The rest of the paper is organized as follows. Section 2 briefly reviews the wide use of PC-based prediction techniques in computer architecture design. Section 3 motivates the pattern classification problem in buffer caching and discusses the design choices. Section 4 presents the PCC design and Section 5 presents the results of an experimental evaluation of PCC. Section 6 discusses additional related work in buffer caching. Finally, Section 7 concludes the paper.

2 PC-based techniques in architecture

History-based prediction techniques exploit the principle that most programs exhibit certain degrees of repetitive behavior. For example, subroutines in an application are called multiple times, and loops are written to process a large amount of data. The challenge in making an accurate prediction is to link the past behavior (event) to

its future reoccurrence. In particular, predictors need the program context of past events so that future events about to occur in the same context can be identified. The more accurate context information the predictor has about the past and future events, the more accurate prediction it can make about future program behavior.

A key observation made in computer architecture is that a particular instruction usually performs a very unique task and seldom changes behavior, and thus program instructions provide a highly effective means of recording the context of program behavior. Since the instructions are uniquely described by their program counters (PCs) which specify the location of the instructions in memory, PCs offer a convenient way of recording the program context.

One of the earliest predictors to take advantage of the information provided by PCs is branch prediction [40]. In fact, branch prediction techniques have been so successful in eliminating latencies associated with branch resolution that they are implemented in every modern processor. The PC of the branch instruction uniquely identifies the branch in the program and is associated with a particular behavior, for example, to take or not to take the branch. Branch prediction techniques correlate the past behavior of a branch instruction and predict its future behavior upon encountering the same instruction.

The success in using the program counter in branch prediction was noticed and the PC information has been widely used in other predictor designs in computer architecture. Numerous PC-based predictors have been proposed to optimize energy [4, 35], cache management [21, 22], and memory prefetching [1, 6, 13, 19, 33, 38]. For example, PCs have been used to accurately predict the instruction behavior in the processor's pipeline which allows the hardware to apply power reduction techniques at the right time to minimize the impact on performance [4, 35]. In Last Touch Predictor [21, 22], PCs are used to predict which data will not be used by the processor again and free up the cache for storing or prefetching more relevant data. In PC-based prefetch predictors [1, 6, 13, 19, 33, 38], a set of memory addresses or patterns are linked to a particular PC and the next set of data is prefetched when that PC is encountered again.

PC-based techniques have also been used to improve processor performance by predicting instruction behavior in the processor pipeline [12, 36] for better utilization of resources with fewer conflicts, as well as to predict data movement in multiprocessors [21, 26] to reduce communication latencies in multiprocessor systems. Most recently, a PC-based technique was proposed to predict disk I/O activities for dynamic power management [15].

Despite their tremendous success in architecture design, PC-based techniques have not been explored in operating systems design. In this paper, we consider the op-

portunity and viability of PC-based prediction techniques in operating systems design. In particular, we consider the buffer cache management problem, which shares common characteristics with hardware cache management as they essentially deal with different levels of the memory hierarchy.

3 Pattern classification in buffer caching

In this section, we motivate the pattern classification problem in buffer caching and discuss various design options.

3.1 Motivation

One of the most important problems in improving file system performance is to design an effective block replacement scheme for the buffer cache. One of the oldest replacement schemes that is yet still widely used is the Least Recently Used (LRU) replacement policy [9]. The effectiveness of LRU comes from the simple yet powerful principle of locality: recently accessed blocks are likely to be accessed again in the near future. Numerous other block replacement schemes based on recency and/or frequency of accesses have been proposed [17, 18, 24, 27, 30, 37]. However, a main drawback of the LRU scheme and all other schemes based on recency and/or frequency of accesses is that they cannot exploit regularities in block accesses such as sequential and looping references [14, 39].

To overcome this drawback, pattern-based buffer replacement schemes [10, 11, 14, 20, 39] have been proposed to exploit the fact that different types of reference patterns usually have different optimal or best known replacement policies. A typical pattern-based buffer cache starts by identifying and classifying reference patterns among accessed disk blocks. It divides up the cache into subcaches, one for blocks belonging to each pattern. Based on the reference pattern, it then applies an optimal or best known replacement policy to each subcache so as to maximize the cache hit ratio for each subcache. In addition, a pattern-based buffer cache needs to dynamically adjust subcache sizes based on the distribution of different types of accesses with the goal of maximizing the overall cache hit ratio. Experimental results [10, 20] have shown that pattern-based schemes can achieve better hit ratios over pure recency/frequency schemes for a mixture of applications.

3.2 Design space

The design space for pattern classification centers around the granularity at which the classifications are being performed. In particular, the access patterns of an application can be classified on a per-application basis, a per-file ba-

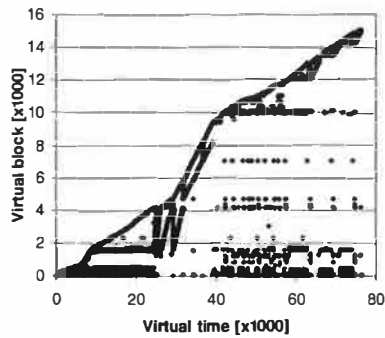
sis, or a per-PC basis, i.e., for each program instruction that triggers any I/O operations.

Per-application classification In a per-application classification scheme such as DEAR [10], the pattern in accesses invoked by the same application is detected and classified. The patterns in DEAR include sequential, looping, temporally clustered, and probabilistic. The scheme periodically reevaluates and reclassifies the patterns and adapts the replacement policy according to the changing reference patterns in the application. However, as shown in [20] and later in this section, many applications access multiple files and exhibit multiple access patterns to different files or even to the same file. Thus the accuracy of per-application classification is limited by its application level granularity.

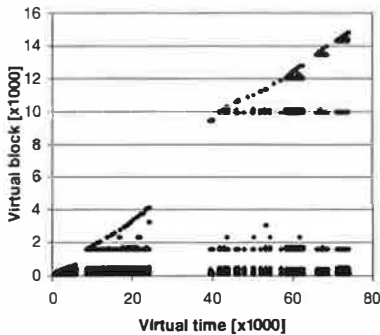
Per-file classification In a per-file classification scheme such as UBM [20], the access pattern in accesses to each file is dynamically classified. In other words, it distinguishes multiple concurrent patterns in the same application as long as they occur in different files. UBM classifies each file into one of the three possible access patterns: sequential, looping, or other. Each file can only have one classification at any given time, though it can be reclassified multiple times. A file receives a sequential classification when some predetermined number (threshold) of consecutive blocks is referenced. Once the file is classified as sequential, it can be reclassified as looping if the file is accessed again according to the sequence seen earlier. The file is classified as having the other reference type when the pattern is not sequential, or the sequence is shorter than the threshold. The Most Recently Used [29] replacement policy is used to manage the subcache for blocks that are accessed with a looping pattern, blocks with a sequential access pattern are not cached, and LRU is used to manage the subcache for blocks of the other reference type.

The classification on a per-file basis in UBM suffers from several drawbacks. First, the pattern detection has to be performed for each new file that the application accesses, resulting in high training overhead and delay in pattern classification when an application accesses a large number of files. Second, since the first iteration of a looping pattern is misclassified as sequential, pattern detection for every new file means such misclassification happens for every file with a looping pattern. Third, if an application performs accesses to a file with mixed access patterns, UBM will not be able to differentiate them. Lastly, the behavior of the threshold-based detection of sequential accesses is directly related to the file size, preventing proper classification of small files.

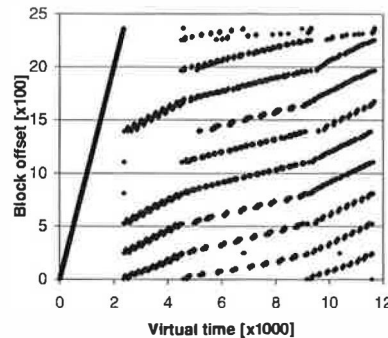
Per-PC classification In a per-PC classification such as PCC proposed in Section 4, the pattern in accesses in-



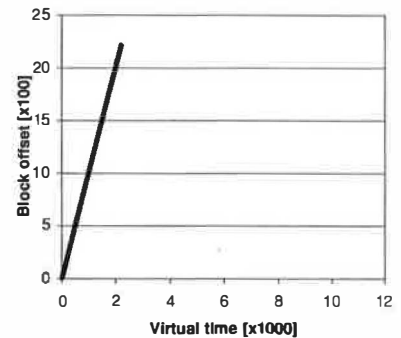
(a) All reference patterns



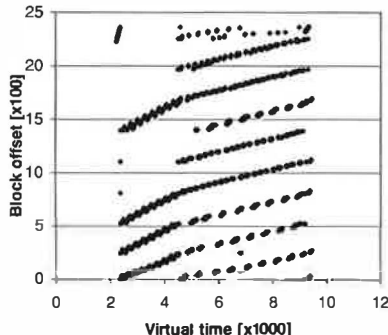
(b) References from reading header files by a single instruction



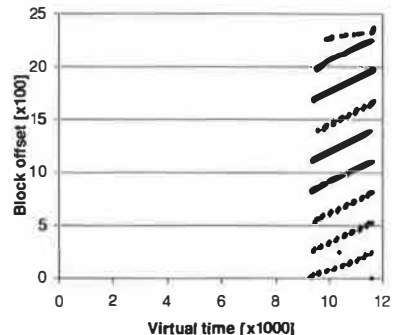
(a) Reference patterns in a single file



(b) Initial scan of the file by a single PC



(c) Processing of the file by multiple PCs



(d) Final references made by a single PC

Figure 2: Reference patterns in *tpc-h*

Figure 1: Reference patterns in *gcc*

voked by a call instruction in the application executable is classified and correlated with future occurrences of the same call instruction (represented by its program counter). Compared to the per-application and per-file classification schemes, per-PC classification correlates the I/O behavior of a program with the program context in which the I/O operations are invoked and records the program context using appropriate PCs. Because of such correlations, per-PC classification is expected to be more accurate than classification schemes that do not exploit such correlations. We illustrate this potential advantage using two of the benchmarks studied in Section 5.

Figure 1(a) shows the space-time graphs of block references in *gcc* (details of which will be discussed in Section 5). The horizontal axis shows the virtual time which is incremented at every block access, and the vertical axis shows the corresponding block number at a given time. Figure 1(a) shows that there exists a mixture of sequential (a single slope) and multiple looping (repeated slopes) patterns in *gcc*. Figure 1(b) presents the reference pattern to blocks accessed by a single instruction in *gcc* responsible for accessing header files during the compilation. Accesses to header files correspond to 67% of the references

in *gcc* of which 99% are reoccurring to header files already accessed once. The remaining 1% of the accesses are the first accesses to repeatedly accessed header files, or to header files that are accessed only once. Most importantly, all header files are accessed by the same single instruction. This observation suggests that the instruction triggering the I/O operations can be used to classify access patterns; the access pattern of the blocks accessed by it needs to be learned once and with high probability, the same pattern holds when it is used to access different files. In contrast, UBM needs to classify the access pattern for each header file, incurring a delay in the classification and, consequently, a missed opportunity in applying the best known replacement scheme.

Figure 2(a) shows the reference patterns to a single file in the *tpc-h* benchmark (details of which will be discussed in Section 5). The vertical axis shows the offset of the blocks in the accessed file. The reference pattern in Figure 2(a) shows a mixture of sequential accesses and looping accesses. To illustrate the use of PCs for pattern classification, we separate the patterns into three different components as shown in Figures 2(b)(c)(d). The accesses in Figure 2(b) are performed by a single PC which

scans the entire file. The blocks read in Figure 2(b) are subsequently accessed again by multiple PCs as shown in Figure 2(c). Since the blocks accessed in Figure 2(b) are accessed again in Figure 2(c), the accessing PC in Figure 2(b) is classified as looping. Similarly, blocks in Figure 2(c) are accessed again in Figure 2(d), and therefore PCs in Figure 2(c) are also classified as looping. Finally, the blocks accessed by a single PC in Figure 2(d) are not accessed again and the accessing PC is classified as sequential.

PCC is able to separate multiple concurrent access patterns and make a proper prediction every time the file with multiple access patterns is referenced. In contrast, UBM classifies access patterns on a per-file basis, and therefore it will not be able to separate the sequential, looping, and other access patterns.

4 PCC Design

The key idea behind PCC design is that *there is a strong correlation between the program context from which I/O operations are invoked and the access pattern among the accessed blocks, and the call instruction that leads to the I/O operations provides an effective means of recording the program context*. Each instruction is uniquely described by its PC. Thus, once PCC has detected a reference pattern, it links the pattern to the PC of the I/O instruction that has performed the accesses. Such pattern classifications are then used by a pattern-based buffer cache to predict access patterns of blocks accessed in the future by the same call instruction.

4.1 Pattern classification

The main task of PCC is to classify the instructions (or their program counters) that invoke I/O operations into appropriate reference pattern categories. Once classified, the classification of the PC is used by the cache manager to manage future blocks accessed by that PC.

We define three basic reference pattern types:

Sequential references are sequences of distinct blocks that are never referenced again;

Looping references are sequential references occurring repeatedly with some interval;

Other references are references not detected as looping or sequential.

Figure 3 shows the two data structures used in PCC implementation. The PC hash table keeps track of how many blocks each PC accesses once (Seq) and how many are accessed more than once (Loop). The block hash table keeps records of M recently referenced blocks, each

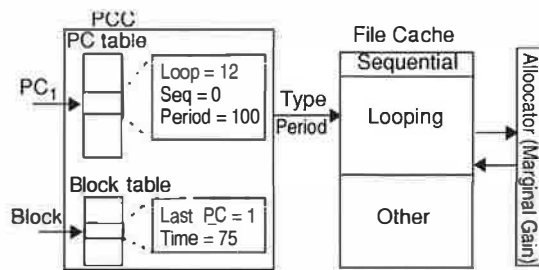


Figure 3: Data structures in PCC

containing the block address, the last accessing PC, and the access time. The choice of the M blocks is discussed in Section 4.2. The access time is simply a count of the number blocks accessed since the program started. PCC maintains the last accessing PC in the block table since a block may be accessed by different PCs.

To simplify the description, we first describe the PCC algorithm assuming all blocks accessed by each PC are monitored. The actual PCC uses a sampling technique to reduce the block hash table size and is discussed in Section 4.2. The pseudo code for PCC without sampling is shown in Figure 4. When a PC triggers an I/O to a block, PCC first looks up the block in the block table to retrieve the PC and the time of the last access to the block. The last accessing PC is used to retrieve the record of that last PC in the PC table. The record is updated by decreasing the Seq count and increasing the Loop count since the block is being accessed again. The exponential average of the period is then calculated based on the time difference recorded in the block entry. If the current PC differs from the last PC, PCC performs an additional lookup into the PC table to obtain the record of the current PC. At this time PCC classifies the reference pattern and returns both the type and the period from the PC entry.

PCC classifies accesses based on the Loop count, the Seq count, and a threshold variable. The threshold aids in the classification of sequential references made by a newly encountered PC. If there are fewer non-repeating (Seq) blocks than repeating (Loop) blocks, the PC is classified as looping, disregarding the threshold. Otherwise, the PC is classified as sequential if the Seq count is larger than or equal to the threshold, or other if the Seq count is smaller than the threshold. PCs that temporarily have comparable Seq and Loop counts and therefore do not fall clearly into sequential or looping categories are classified as other references. These PCs mostly have a combination of sequential references to some blocks and looping references to other blocks.

Similar to UBM, PCC will misclassify the first occurrence of a looping sequence as sequential, assuming it is longer than the threshold. However, once PCC assigns the looping pattern to a PC, the first references to any file by

```

PCC(PC, Block, currTime)
(1) if ((currBlock = getBlockEntry(Block)) == NULL)
(2)   currBlock = NewBlockEntry(Block);
(3) else { // the entry for last accessing PC must exist
(4)   currPC = getPCEntry(currBlock->Last_PC);
(5)   currPC->Seq- -;
(6)   currPC->Loop++;
(7)   currPC->Period = expAverage(currPC->Period,
(8)     currTime - currBlock->Time);
(9) }
(10) if (currBlock->Last_PC != PC)
(11)   currPC = getPCEntry(PC);
(12) if (currPC == NULL) {
(13)   currPC = NewPCEntry(PC);
(14)   currPC->Seq = 1;
(15)   currPC->Loop = currPC->Period = 0;
(16)   Type = "Other";
(17) } else {
(18)   currPC->Seq++;
(19)   if (currPC->Loop > currPC->Seq)
(20)     Type = "Looping";
(21)   else if (currPC->Seq >= Threshold)
(22)     Type = "Sequential";
(23)   else
(24)     Type = "Other";
(25)   Period = currPC->period;
(26) }
(27) currBlock->Time = currTime;
(29) currBlock->Last_PC = PC;
(30) return(Type, Period);

```

Figure 4: Pseudocode for PCC without sampling

the same PC will be labeled as looping right away.

We note the threshold value has different meaning and performance impact in PCC and UBM. In UBM, the threshold value is on a per-file basis and therefore files with sizes smaller than the threshold are not classified properly. In PCC, the threshold is set on a per-PC basis, and thus a PC that accesses enough small files will be properly classified.

Figure 5 shows some example reference patterns assuming that the threshold is set at three. When the application starts executing, PCC observes a set of sequential references by PC1. After three initial references are classified as other, the threshold is reached and PC1 is classified as sequential for the remaining references. When the sequence is accessed again, PCC reclassifies PC1 as looping, and future blocks referenced by PC1 are classified as looping. At that time, the loop period is calculated and recorded with PC1. In the meantime, PC2 is encountered and again classified after the initial three references as sequential. The classification is not changed for PC2 since no looping references are encountered. When PC3 accesses the same set of blocks that were accessed by PC1, PC3 is classified as sequential, since PC3 is observed for the first time. Note that the classification of PC1 remains

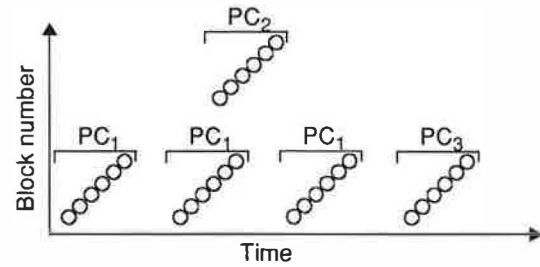


Figure 5: An example of reference patterns

the same, although the classification of the set of blocks it accessed has changed from looping to sequential.

4.2 PCC with block sampling

Maintaining the blocks in the block hash table is crucial to calculating the loop period. Keeping the information for every accessed block would be prohibitive for large data sets. Instead, PCC only keeps periodically sampled referenced blocks in the block hash table, and repeated accesses to the sampled blocks are used to calculate the loop period for each PC. Specifically, for each PC in the application, the block hash table maintains up to N blocks, each recorded at every T -th access by that PC. We discuss the choices of N and the sampling period T in Section 5.3.1. Note that sampling dictates which blocks will be inserted in the block table and thus used for calculating the average period for a PC. If the block in the block table is accessed again, the PC period calculation is performed as before. After the update, the block is discarded from the block table to free a slot for the next sample.

To further limit the block table size, when the block table is full, PCC uses the LRU policy to evict the least recently used PC from the PC table and the corresponding block table entries.

4.3 Obtaining signature PCs

Instead of obtaining a single PC of the function call from the application that invokes each I/O operation, PCC actually uses a *signature PC* which records the *call site* of the I/O operation by summing the sequence of PCs encountered in going through multiple levels of wrappers before reaching the actual system call. The wrapper functions are commonly used to simplify programming by abstracting the details of accessing a particular file structure. For example, in the call graph shown in Figure 6, Functions 2, 3, and 4 use the same PC in the wrapper function for I/O operations. Therefore, the PC that invokes the I/O system call within the wrapper cannot differentiate the behavior of different caller functions. To obtain a unique characterization of the accessing PC, PCC traverses multiple function stacks in the application. The PCs obtained during the stack frame traversal are summed

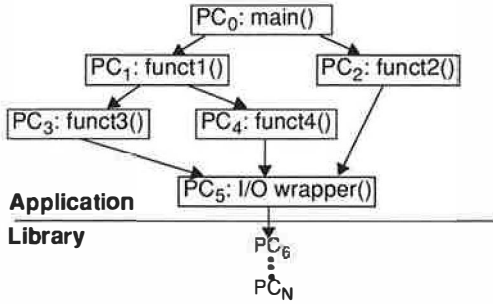


Figure 6: An example function call graph

together to obtain a unique identifier as the *signature PC* of the I/O operation. In the applications studied in Section 5, traversal of only two additional stack frames provided sufficient information to PCC. As the minimum number of stack frames needed to generate unique signature PCs varies from application to application, PCC always traverses all the function stacks in the application until reaching `main()`. We note that it is extremely unlikely that the signature PCs of different call sites will collide as each signature PC is 32-bit, while an application typically has up to a few hundred call sites. For simplicity, in the rest of the paper, we will refer to the signature PC of an I/O operation simply as its PC.

Compared to previous pattern classification schemes, a unique property of PCC is that the PCs that trigger I/O operations and their access patterns tend not to change in future invocations of the same application. Thus, PCC saves the per-PC pattern classifications in the PC table for potential future reuse. Similar to the block table, the size of the PC table can be easily managed using LRU. We note that the PC table is usually much smaller than the block table as most applications have a few call sites responsible for accessing a large number of blocks.

5 Performance evaluation

We compare the performance of UBM, PCC, ARC [27], and LRU via trace-driven simulations and an actual implementation in Linux. Simulated UBM results were obtained using the unmodified UBM simulator from the authors of UBM [20]. To obtain results for PCC, we modified the UBM simulator by replacing its classification routine with PCC; UBM’s cache management based on marginal gain evaluations was kept without any modification. Using the same marginal gain in PCC and UBM isolates the performance difference due to different pattern classification schemes. We also implemented the ARC scheme [27] as presented in [28]. ARC is a state-of-the-art recency/frequency-based policy that offers comparable performance to the best online and off-line algorithms. It dynamically adjusts the balance between the LRU and LFU components for a changing workload by maintain-

ing two LRU lists: one contains pages seen only once while the other contains pages seen more than once. At any given time, ARC selects top elements from both lists to reside in the cache. Finally, we implemented both PCC and UBM in Linux kernel 2.4.

5.1 Cache organization and management

Both the UBM-based and the PCC-based replacement schemes in our comparison study use the marginal gain in the original UBM [20] to manage the three partitions of the buffer cache, used to keep blocks with sequential, looping, and other references, respectively. Once the blocks are classified, they are stored in the appropriate subcache and managed with the corresponding replacement policy. Sequentially referenced blocks, as defined, are not accessed again, and therefore they can be discarded immediately after being accessed. Looping references are primarily managed based on the looping interval. Looping blocks with the largest interval will be replaced first since they will be used furthest in the future. If all blocks in the cache have the same detected interval, the replacement is made based on the MRU [29] replacement policy. References classified as other are managed by LRU as in the original UBM [20], but can be managed by other recency/frequency-based policies as well.

The cache management module uses marginal gain computation to dynamically allocate the cache space among the three reference types [29, 43]. As mentioned earlier, sequential references can be discarded immediately. Since there is no benefit from caching them, the marginal gain is zero, and the subcache for sequential references consists of only one block per application thread. The remaining portion of the cache is distributed dynamically between sequential and other references. The benefit of having an additional block is estimated for each subcache, and the block is removed from the subcache that would have gained less by having the additional block and given to the subcache that will benefit more.

5.2 Applications

Tables 1 and 2 show the five applications and three concurrent executions of the mixed applications used in this study. For each application, Table 1 lists the number of I/O references, the size of the I/O reference stream, the number of unique files accessed, and the number of unique signature PCs used for the I/O references. The selected applications and workload sizes are comparable to the workloads in recent studies [11, 17, 24, 27] and require cache sizes up to 1024MB.

Cscope [42] performs source code examination. The examined source code is Linux kernel 2.4.20. *Glimpse* [25] is an indexing and query system and is used

Appl.	Num. of references	Data size [MB]	Num. of files	Num. of PCs
<i>cscope</i>	1119161	260	10635	107
<i>glimpse</i>	519382	669	43649	25
<i>gcc</i>	158667	41	2098	334
<i>viewperf</i>	303123	495	289	179
<i>tpc-h</i>	121307	196	80	242
<i>multi1</i>	1278135	297	12246	442
<i>multi2</i>	1580908	792	12514	605
<i>multi3</i>	640467	865	43738	268

Table 1: Applications and trace statistics

Appl.	Applications executed concurrently
<i>multi1</i>	<i>cscope</i> , <i>gcc</i>
<i>multi2</i>	<i>cscope</i> , <i>gcc</i> , <i>viewperf</i>
<i>multi3</i>	<i>glimpse</i> , <i>tpc-h</i>

Table 2: Concurrent applications

to search for text strings in 550MB of text files under the `/usr` directory. In both *Cscope* and *Glimpse*, an index is first built, and single word queries are then issued. Only I/O operations during the query phases are used in the experiments. In both applications, looping references dominate sequential and other references. *Gcc* builds Linux kernel 2.4.20 and is one of the commonly used benchmarks. It shows both looping and sequential references, but looping references, i.e., repeated accesses to small header files, dominate sequential references. As a result, it has a very small working set; 4MB of buffer cache is enough to contain the header files. This constrains the buffer cache sizes used in the evaluation. *Viewperf* is a *SPEC* benchmark that measures the performance of a graphics workstation. The benchmark executes multiple tests to stress different capabilities of the system. The patterns are mostly regular loops as *viewperf* reads entire files to render images. The *Postgres* [34] database system from the University of California is used to run *TPC-H* (*tpc-h*) [44]. *Tpc-h* accesses a few large data files, some of which have multiple concurrent access patterns.

Multi1 consists of concurrent executions of *cscope* and *gcc*. It represents the workload in a code development environment. *Multi2* consists of concurrent executions of *cscope*, *gcc*, and *viewperf*. It represents the workload in a workstation environment used to develop graphical applications and simulations. *Multi3* consists of concurrent executions of *glimpse* and *tpc-h*. It represents the workload in a server environment running a database server and a web index server.

We briefly discuss the characteristics of the individual applications that affect the performance of the eviction policies. Figure 7 shows the cumulative distribution of the references to the files in these applications. We observe that the number of files contributing to the number

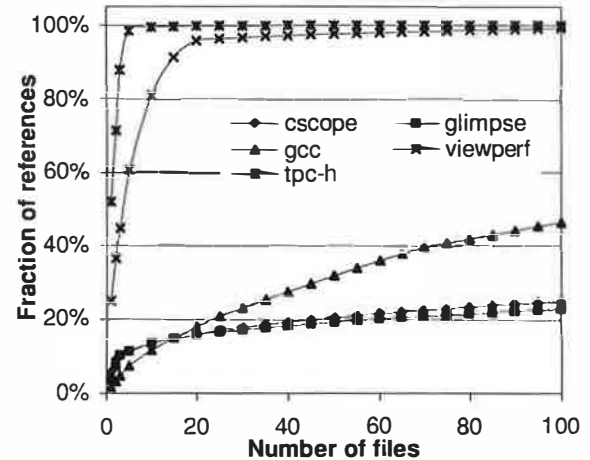


Figure 7: Cumulative distribution of file accesses

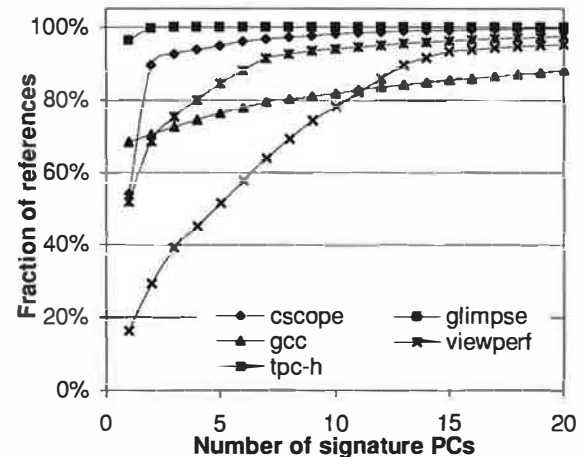


Figure 8: Cumulative distribution of signature PCs

of references spans a wide range. The number of files affects the performance of UBM. Since UBM trains on each new file to detect the looping and sequential references, the amount of training in UBM is proportional to the number of files that an application accesses. The number of I/O call sites (signature PCs) has a similar impact on training in PCC. Figure 8 shows the cumulative distribution of the references triggered by signature PCs in the applications. Compared to UBM, in PCC, fewer than 30 PCs are responsible for almost all references in all applications, resulting in shorter training in PCC than in UBM. Lastly, the average number of application function stack frame traversals to reach `main()` for the eight application versions is 6.45.

5.3 Simulation results

The detailed traces of the applications were obtained by modifying the *strace* Linux utility. *Strace* intercepts the system calls of the traced process and is modified to

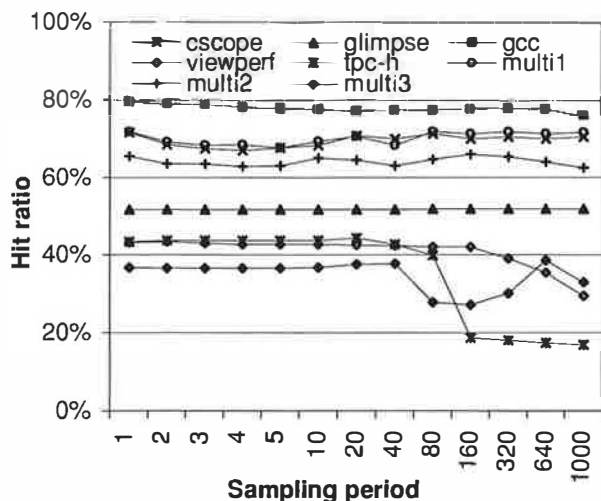


Figure 9: Impact of sampling period on PCC's hit ratio

record the following information about the I/O operations: PC of the calling instruction, access type, file identifier (inode), and I/O size.

5.3.1 Impact of sampling frequency in PCC

Figure 9 shows the impact of block sampling on the hit ratio in PCC. We selected a single cache size for each benchmark – 128MB for *cscope*, *tpc-h*, *multi1*, and *multi2*, 512MB for *glimpse* and *multi3*, 2MB for *gcc*, and 32MB for *viewperf*. The same cache sizes are used in Section 5.4 to compare the performance of LRU, UBM, and PCC in our Linux implementation. No limit is imposed on the block table size, and the threshold value is set to 100. Figure 9 shows that increasing the sampling period to 20 barely affects the hit ratio for all applications. However, as the sampling period continues to increase, PCC may not capture changes in the loop periods and result in reduced hit ratios. We also performed sensitivity analysis on the threshold value of PCC. The results show that varying the threshold between 5 and 400 results in less than a 2% variation in hit ratios for the eight application versions. Thus, we chose a sampling period of 20 and a threshold of 100 in our simulation and implementation experiments below.

Using sampling significantly reduces the storage overhead of the block table. Recording all blocks accessed by an application require as much as 220K entries in the block table for the eight application versions in Table 1. Using a sampling period of 20, the maximum number of entries in the block table is less than 9K entries for all applications. Since the memory overhead of a block table with 9K entries is comparable to that of the per-file pattern table in UBM (up to 20K entries for these applications), we do not explore the impact of the LRU replacement in the PC table in this paper.

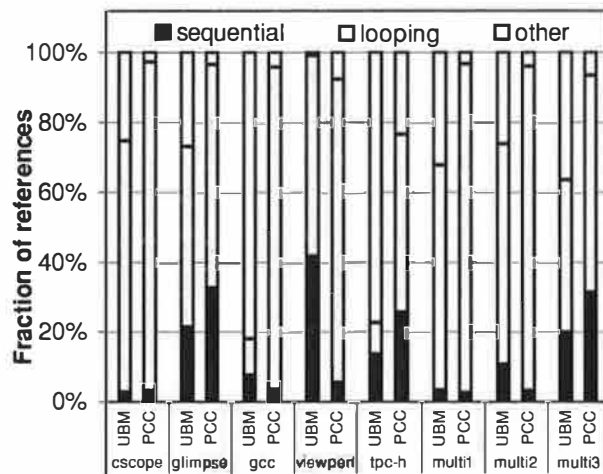


Figure 10: Reference classification in UBM and PCC

5.3.2 Pattern classification

To help understand the measured hit ratios, we first present the classification results for PCC and UBM in Figure 10. The classification results are a characteristic of the applications and are independent of the cache size. The accuracies of the classification schemes will dictate how often the cache manager can apply the appropriate eviction policies for the accessed blocks.

UBM and PCC on average have 15% and 13% sequential references, respectively. PCC has an advantage in detecting looping references when the same PC is used to access multiple files. As a result, PCC has an average of 80% looping references and only 7% other references. UBM classifies many more references as other, resulting in an average 46% looping and 39% other references. In the cases of *gcc* and *tpc-h*, UBM classifies a limited fraction of references, and thus the performance of UBM is expected to degrade to that of LRU. In these cases, ARC and PCC will provide improved performance over UBM.

5.3.3 Cache hit ratios

Figure 11 shows the hit ratios for the studied applications under different cache sizes in Megabytes. We plot the results for five replacement policies for each application: OPT, PCC, UBM, ARC, and LRU. The optimal replacement policy (OPT) assumes future knowledge and selects the cache block for eviction that is accessed furthest in the future [3]. PCC performs block sampling as described in Section 4.2. The threshold value of 3 was found to be optimal for UBM for the studied applications and was used in the experiments.

Overall, compared to UBM, PCC improves the absolute hit ratio by as much as 29.3% with an average of 13.8% maximum improvement over the eight application versions. Compared to ARC, PCC improves the absolute

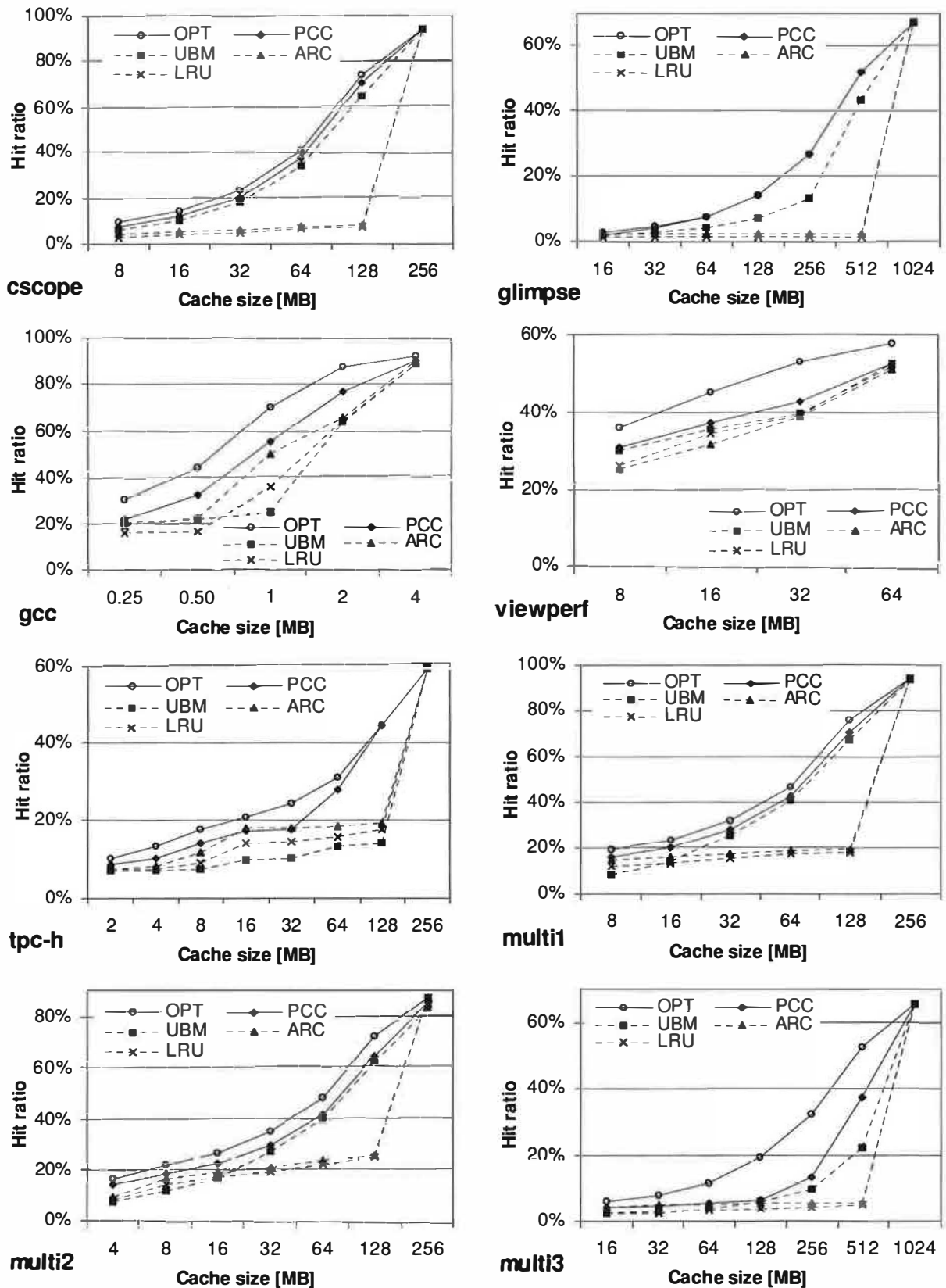


Figure 11: Comparison of cache replacement schemes

hit ratio by as much as 63.4% with an average of 35.2% maximum improvement.

Cscope *Cscope* scans an index file and many text files during source code examination and shows the looping behavior for a large number of files. This is a pathological case for LRU when the blocks accessed in the loop cannot fit in the cache. Similar behavior is present in ARC as all blocks are accessed more than once and the frequency list is also managed by LRU. The benefit of LRU and ARC caching is only observed when the entire looping file set fits in the cache. Both UBM and PCC show significant gains over LRU and ARC. UBM is able to classify the references and take advantage of the MRU replacement policy, achieving a maximum of 57.3% improvement in hit ratio over LRU. In contrast, PCC achieves as much as a 64.4% higher hit ratio than LRU and 7.1% higher than UBM. Two factors contribute to the hit ratio difference between PCC and UBM. First, UBM is unable to classify small files which are inside a loop and account for 10% of the references. Second, *Cscope* periodically calls the seek function to adjust the file position during sequential reading of the index file. The adjustment, although very small, disrupts the looping classification in UBM, causing the the classification to switch to other and subsequently to sequential. In contrast, PCC only observes small fluctuations in the resulting loop period but maintains the looping classification.

Glimpse *Glimpse* is similar to *cscope* as it scans both an index file and many text files during string search. UBM has to train for each of the 43649 files it accesses. It classifies 56% of these files as other since they have sizes below the threshold of three blocks. Each of the remaining files is first classified as other during the initial two references, and then as sequential upon the third reference, and finally looping once a loop is detected. Since all of these files are accessed by a single PC, PCC will train much more quickly, resulting in the accurate classification of looping. Figure 10 shows that PCC detects 12% more looping references than UBM. Figure 11 shows that PCC closely matches the hit ratio of OPT across different cache sizes as it is able to classify all references as looping and apply the MRU policy to the accessed blocks.

Gcc In *Gcc*, 50% of references are to files shorter than the threshold. As discussed in Section 3.2, accesses to all the headers files are triggered by a single PC, and thus PCC is able to learn the looping access pattern of the PC once and makes a correct looping prediction for all the remaining accesses to the header files. In contrast, files shorter than the threshold will be classified by UBM as having other references. Figure 10 shows PCC is able to detect more looping references as compared to UBM, and Figure 11 shows PCC achieves a signifi-

cantly higher hit ratio than UBM. When the cache size is 1MB, UBM achieves a lower hit ratios than LRU. This is due to accesses to temporary files that are accessed only twice. First-time accesses to a temporary file are misclassified by UBM as sequential and discarded from the cache. Second-time accesses to the temporary file are again misclassified by UBM as looping and the blocks are placed in the loop cache, taking resources away from the LRU cache. These misclassifications in UBM hurt the hit ratio the most when the cache size is 1MB.

Viewperf *Viewperf* is driven by five different datasets, each having several files describing the data. Over 95% of the references when driven by the five datasets are to 8, 4, 1, 4, and 3 different files, respectively. Each file is reused as input to several invocations of the viewperf application with different parameters and is then discarded. A total of four files in *viewperf* are accessed only twice, which results in wrong classifications in UBM for both passes, similarly as in *gcc*. As in *gcc*, PCC uses only the first file to learn and correctly classifies accesses to the remaining files, resulting in higher hit ratios than UBM.

Tpc-h *Tpc-h* has three files that account for 88% of all accesses, and therefore the training overhead for new files is not a major issue for UBM. The main difference between PCC and UBM is due to accesses to one of the three files which accounts for 19% of the references. As explained in Section 3, accesses to the file contain multiple concurrent looping references, and UBM will misclassify all references to the file. Accesses in Figures 2(b) and 2(c) should be classified as looping, and accesses in Figure 2(d) sequential. However, irregularities and multiple concurrent looping references will cause UBM to classify accesses in Figures 2(b)(c)(d) as other. Because of the high percentage of other references in UBM as shown in Figure 10, UBM performs similarly as LRU as shown in Figure 11.

Multi1, multi2 and multi3 The traces for *multi1*, *multi2*, and *multi3* contain the references of the individual applications and thus inherit their characteristics. The pattern detection is performed as in the case of individual applications, since there is no interference among different applications due to their distinct PCs.

The resulting cache hit ratio curves are dominated by the applications that perform accesses more frequently, i.e., *cscope* in *multi1* and *multi2* and *glimpse* in *multi3*. In *multi1* and *multi2*, *gcc* is able to cache the looping header files in under 4MB, while for larger buffer sizes, the access pattern is primarily dominated by the remaining applications. Interleaved references from *glimpse* and *tpc-h* in *multi3* affect the period calculation, resulting in lower hit ratios for PCC than expected from individual executions of the two applications as shown in Figure 11.

5.4 Implementation in Linux

We implemented both UBM-based and PCC-based replacement schemes in Linux kernel 2.4.20 running on a 2Ghz Intel Pentium 4 PC with 2GB RAM and a 40GB Seagate Barracuda hard disk. We modified the kernel to obtain the signature PCs. Specifically, we modified the `read` and `write` system calls such that upon each access to these calls, PCC traverses the call stack to retrieve the relevant program counters. Since multiple levels of stacks have to be traversed to determine the signature PC, the traversal involves repeatedly accessing the user space from the kernel space.

We also modified the cache in Linux to allow setting a fixed cache size. This modification allows us to vary the cache size and study the effectiveness of the classification schemes for different cache sizes while keeping all other system parameters unchanged. One limitation of our implementation is that it cannot retrieve the PCs for references to files mapped into memory by `mmap`. Such references are classified as other references. Out of the applications we studied, only `gcc` has a few memory mapped file accesses.

Based on the working set sizes, we selected the cache size to be 128MB for `cscope`, `tpc-h`, `multi1`, and `multi2`, 512MB for `glimpse` and `multi3`, 2MB for `gcc`, and 32MB for `viewperf`. Figure 12 shows the number of disk I/Os and the execution time for each application under UBM and PCC normalized to those under the basic LRU scheme. For all three schemes, the standard file system prefetching of Linux (`generic_file_readahead`) was enabled, which prefetches up to 32 blocks from the disk for sequential accesses. The execution times reported here are the average of three separate executions. Each execution was run after the system was rebooted to eliminate any effects from prior buffer cache content.

Overall, compared to the basic LRU, UBM reduces the average number of disk I/Os by 20.7%, while better classification in PCC results in an average of 41.5% reduction in the number of disk I/Os, or 20.8% over UBM. The reduction in disk I/Os leads to a reduction in the execution time; UBM reduces the average execution time of LRU by 6.8%, while PCC reduces the average execution time of LRU by 20.5%, or 13.7% compared to UBM. We notice that in `viewperf` the small decrease in I/Os does not translate into saving in execution time because `viewperf` is a CPU-bound application. All other applications are I/O-bound and the reduction in execution time follows the reduction in the number of I/Os.

5.4.1 Runtime overhead of PCC

To measure the overhead of PCC, we used a microbenchmark that repeatedly reads the same block of a file which results in hits under all of the LRU, UBM, and PCC

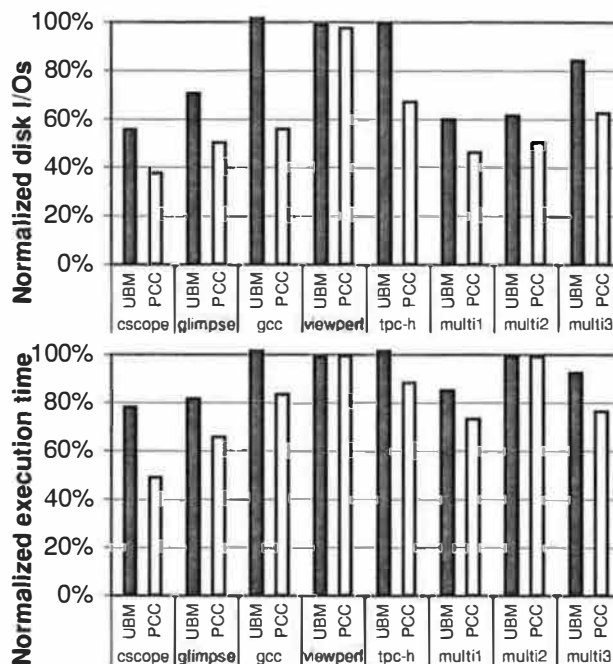


Figure 12: Performance of UBM and PCC integrated into the Linux kernel (Normalized to the performance of the basic LRU scheme)

schemes. Since the average number of application stack frame traversals in our experiments is 6.45, we set our microbenchmark to go through 7 subroutine calls before invoking the actual system call. The time to service a hit in the buffer cache in the unmodified kernel is 2.99 microseconds. In our implementation, this time increases to 3.52 microseconds for LRU because of code changes we incorporated to support a fixed cache size and different replacement schemes. The time to service a hit in UBM is 3.56 microseconds due to the marginal gain cache management and pattern classification overhead. This time increases to 3.75 microseconds in PCC. The additional overhead of 0.19 microsecond is due to obtaining the signature PC of the I/O operation. The additional overhead of 0.23 microsecond compared to LRU is promising: one saved cache miss which would cost 5-10 milliseconds is equivalent to the PCC overhead in about 20000 – 40000 cache accesses (hits or misses), and thus the overhead is expected to be overshadowed by the gain from the improved cache hit ratio.

Finally, we measured the impact of PCC overhead when the applications do not benefit from PCC. Using a 1GB cache, which is large enough that all replacement schemes result in identical cache hit ratios for all five individual applications. The average running time of the five applications using PCC is 0.65% longer than that using LRU.

6 Related work

The vast amount of work on PC-based techniques in computer architecture design have been discussed in Section 2. Here we briefly review previous buffer caching replacement policies which fall into three categories: recency/frequency-based, hint-based, and pattern-based.

Frequency/Recency-based policies Despite its simplicity, LRU can suffer from its pathological case when the working set size is larger than the cache and the application has looping access pattern. Similarly, LFU suffers from its pathological case when initially popular cache blocks are subsequently never used. Many policies have been proposed to avoid the pathological cases of LRU and LFU. LRU-K [30, 31] is related to LFU and replaces a block based on the K th-to-the-last reference. LRU-K is more adaptable to the changing behavior but it still requires the logarithmic complexity of LFU to manage the priority queue. To eliminate the logarithmic complexity of LRU-K, 2Q [18] maintains two queues: one queue for blocks referenced only once, and another for reoccurring references. If a block in the first queue is referenced again, it is moved to the second queue. This simple algorithm results in constant complexity per access; however, it requires two tunable parameters. Low Inter-reference Recency Set (LIRS) [17] maintains a complexity similar to that of LRU by using the distance between the last and second-to-the-last references to estimate the likelihood of the block being referenced again.

Many policies have been proposed to combine recency and frequency. The first policy to combine LRU and LFU is Frequency-Based Replacement (FBR) [37]. It combines the access frequency with the block age by maintaining an LRU queue divided into three sections: new, middle, and old. Least Recently/Frequently Used (LRFU) [24] provides a continuous range of policies between LRU and LFU. A parameter λ is used to control the amount of recency and frequency that is included in the value used for replacement. Adaptive LRFU (ALRFU) [23] dynamically adjusts λ , eliminating the need to properly set λ for a particular workload. The most recent additions to the LFU/LRU policies are Adaptive Replacement Cache (ARC) [27] and its variant CAR [2]. The basic idea of ARC/CAR is to partition the cache into two queues, each managed using either LRU (ARC) or CLOCK (CAR): one contains pages accessed only once, while the other contains pages accessed more than once. Like LRU, ARC/CAR has constant complexity per request.

Hint-based policies In application controlled cache management [8, 32], the programmer is responsible for inserting hints into the application which indicate to OS what data will or will not be accessed in the future and when. The OS then takes advantage of these hints to de-

cide what cached data to discard and when. This can be a difficult task as the programmer has to carefully consider the access patterns of the application so that the resulting hints do not degrade the performance. To eliminate the burden on the programmers, compiler inserted hints were proposed [7]. These methods provide the benefits of user inserted hints for existing applications that can be simply recompiled with the proposed compiler. However, more complicated access patterns or input dependent patterns may be difficult for the compiler to characterize.

Pattern-based policies Dynamically adaptable pattern detection not only eliminates the burden on the programmer but also adapts to the user behavior. SEQ [14] detects sequential page fault patterns and applies the Most Recently Used (MRU) policy to those pages. For other pages, the LRU replacement is applied. However, SEQ does not distinguish sequential and looping references. EELRU [39] detects looping references by examining aggregate recency distribution of referenced pages and changes the eviction point using a simple cost/benefit analysis. As discussed in Section 3, DEAR and UBM [10, 11, 20] are closely related to PCC in that they are pattern-based buffer cache replacement schemes and explicitly separate and manage blocks that belong to different reference patterns. They differ from PCC in the granularity of classification; classification is on a per-application basis in DEAR, a per-file basis in UBM, and a per-call-site basis in PCC.

7 Conclusions

This paper presents the first design that applies PC-based prediction to the I/O management in operating systems. The proposed PCC pattern classification scheme allows the operating system to correlate I/O operations with the program context in which they are triggered, and thus has the potential to predict access patterns much more accurately than previous schemes. Compared to the per-file access pattern classification scheme in UBM, PCC offers several advantages: (1) it can accurately predict the reference patterns of new files before any access is performed, eliminating the training delay; (2) it can differentiate multiple concurrent access patterns in a single file.

Our evaluation using a range of benchmarks shows that, compared to UBM, PCC achieves on average a 13.8% higher maximum hit ratio in simulations with varying cache sizes, and reduces the average number of disk I/Os by 20.8% and the average execution time by 13.7% in our Linux implementation. These results demonstrate that exploiting the synergy between architecture and operating system techniques to solve problems of common characteristics, such as exploiting the memory hierarchy, is a promising research direction.

Acknowledgments

We wish to thank Mendel Rosenblum, our shepherd, and the anonymous reviewers for their constructive feedback on this work. We would also like to thank Jongmoo Choi for giving us access to the UBM simulator code.

References

- [1] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proc. ICS*, June 1991.
- [2] S. Bansal and D. S. Modha. CAR: Clock with Adaptive Replacement. *Proc. FAST*, March 2004.
- [3] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.
- [4] N. Bellas, I. Hajj, and C. Polychronopoulos. Using dynamic cache management techniques to reduce energy in a high-performance processor. In *Proc. ISLPED*, August 1999.
- [5] D. Bitton, D. DeWitt, and C. Turbyfill. Benchmarking database systems: A systematic approach. In *Proc. VLDB*, October 1983.
- [6] B. Black, B. Mueller, S. Postal, R. Rakvic, N. Utamaphethai, and J. P. Shen. Load execution latency reduction. In *Proc. ICS*, July 1998.
- [7] A. D. Brown, T. C. Mowry, and O. Krieger. Compiler-based I/O prefetching for out-of-core applications. *ACM TOCS*, 19(2):111–170, 2001.
- [8] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM TOCS*, 14(4):311–343, 1996.
- [9] R. W. Carr and J. L. Hennessy. WSCLOCK – a simple and effective algorithm for virtual memory management. In *Proc. SOSP-8*, December 1981.
- [10] J. Choi, S. H. Noh, S. L. Min, and Y. Cho. An Implementation study of a detection-based adaptive block replacement scheme. In *Proc. 1999 USENIX ATC*, June 1999.
- [11] J. Choi, S. H. Noh, S. L. Min, and Y. Cho. Towards application/file-level characterization of block references: a case for fine-grained buffer management. In *Proc. ACM SIGMETRICS*, June 2000.
- [12] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proc. ISCA*, June 1998.
- [13] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *Proc. ISCA*, June 1997.
- [14] G. Glass and P. Cao. Adaptive page replacement based on memory reference behavior. In *Proc. ACM SIGMETRICS*, June 1997.
- [15] C. Gniady, Y. C. Hu, and Y.-H. Lu. Program counter based techniques for dynamic power management. In *Proc. HPCA-10*, February 2004.
- [16] R. J. Hanson. TPC Benchmark B - What it means and how to use it. In *Transaction Processing Performance Council. TPC-B standard specification, revision 2.0*, 1994.
- [17] S. Jiang and X. Zhang. LJRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proc. ACM SIGMETRICS*, June 2002.
- [18] T. Johnson and D. Shasha. 2Q: a low overhead high performance buffer management replacement algorithm. In *Proc. VLDB-20*, September 1994.
- [19] Y. Jgou and O. Temam. Speculative prefetching. In *Proc. ICS-7*, July 1993.
- [20] J. M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. A low-overhead, high-performance unified buffer management scheme that exploits sequential and looping references. In *Proc. OSDI*, October 2000.
- [21] A.-C. Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proc. ISCA*, June 2000.
- [22] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction and dead-block correlating prefetchers. In *Proc. ISCA*, June 2001.
- [23] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *Proc. ACM SIGMETRICS*, May 1999.
- [24] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, 50(12):1352–1360, 2001.
- [25] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Proc. USENIX Winter 1994 Technical Conference*, January 1994.
- [26] M. M. K. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood. Using destination-set prediction to improve the latency/bandwidth tradeoff in shared-memory multiprocessors. In *Proc. ISCA*, June 2003.
- [27] N. Megiddo and D. S. Modha. ARC: A Self-tuning, Low Overhead Replacement Cache. In *Proc. FAST*, March 2003.
- [28] N. Megiddo and D. S. Modha. One up on LRU. *login: - The Magazine of the USENIX Association*, 18(4):7–11, 2003.
- [29] R. Ng, C. Faloutsos, and T. Sellis. Flexible and adaptable buffer management techniques for database management systems. *IEEE Transactions on Computers*, 44(4):546–560, 1995.
- [30] E. J. O’Neil, P. E. O’Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proc. ACM SIGMOD*, May 1993.
- [31] E. J. O’Neil, P. E. O’Neil, and G. Weikum. An optimality proof of the LRU-K page replacement algorithm. *J. ACM*, 46(1):92–112, 1999.
- [32] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proc. SOSP-15*, December 1995.
- [33] S. S. Pinter and A. Yoaz. Tango: a hardware-based data prefetching technique for superscalar processors. In *Proc. MICRO-29*, December 1996.
- [34] Postgres. <http://www.postgresql.org/>.
- [35] M. D. Powell, A. Agarwal, T. N. Vijaykumar, B. Falsafi, and K. Roy. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In *Proc. MICRO-34*, Dec. 2001.
- [36] G. Reinman and B. Calder. Predictive techniques for aggressive load speculation. In *Proc. MICRO-31*, December 1998.
- [37] J. T. Robinson and M. V. Devarakonda. Data cache management using frequency-based replacement. In *Proc. ACM SIGMETRICS*, May 1990.
- [38] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *Proc. ISCA*, June 2000.
- [39] Y. Smaragdakis, S. Kaplan, and P. Wilson. EELRU: simple and effective adaptive page replacement. In *Proc. ACM SIGMETRICS*, May 1999.
- [40] J. E. Smith. A study of branch prediction strategies. In *Proc. ISCA*, May 1981.
- [41] SPEC. <http://www.spec.org/gpc/opc.static/viewperf71info.html>.
- [42] J. Steffen. Interactive examination of a C program with Cscope. In *Proc. USENIX Winter 1985 Technical Conference*, 1985.
- [43] D. Thibaut, H. S. Stone, and J. L. Wolf. Improving disk cache hit-ratios through cache partitioning. *IEEE Transactions on Computers*, 41(6):665–676, 1992.
- [44] TPC. Transaction Processing Council. <http://www.tpc.org>.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

Member Benefits

- Free subscription to *login:*, the Association's magazine, published six times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see <http://www.usenix.org/membership/specialdisc.html> for details.

SAGE

SAGE is a Special Interest Group (SIG) of the USENIX Association. It is organized to advance the status of computer system administration as a profession, establish standards of professional excellence and recognize those who attain them, develop guidelines for improving the technical and managerial capabilities of members of the profession, and promote activities that advance the state of the art or the community.

USENIX & SAGE Thank Their Supporting Members

USENIX Supporting Members

- ❖ Addison-Wesley/Prentice Hall PTR ❖ Ajava Systems, Inc. ❖ AMD ❖ Asian Development Bank ❖
- ❖ Atos Origin BV ❖ Delmar Learning ❖ DoCoMo Communications Laboratories USA, Inc. ❖
- ❖ Electronic Frontier Foundation ❖ Hewlett-Packard ❖ IBM ❖ Interhack ❖ MacConnection ❖
- ❖ The Measurement Factory ❖ Microsoft Research ❖ Oracle ❖ OSDL ❖ Perfect Order ❖
- ❖ Portlock Software ❖ Raytheon ❖ Sun Microsystems, Inc. ❖ Taos ❖ Tellme Networks ❖
- ❖ UUNET Technologies, Inc. ❖ Veritas Software ❖

SAGE Supporting Members

- ❖ Addison-Wesley/Prentice Hall PTR ❖ Ajava Systems, Inc. ❖ Asian Development Bank ❖
- ❖ Fotosearch ❖ Microsoft Research ❖ MSB Associates ❖ Raytheon ❖
- ❖ Ripe NCC ❖ Taos ❖ Tellme Networks ❖

For more information about membership, conferences, or publications,
see <http://www.usenix.org/>

or contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 Fax: 510-548-5738 Email: office@usenix.org

ISBN 1-931971-26-9